TABLE OF CONTENTS

QEMU/KVM Security Matters

1-) Introduction

This document describes how QEMU works, pointing out how its behavior can be analyzed, audited and logged having in mind execution escapes and privilege escalations.

Before moving into virtual machines logs and confinement options, it is imperative that **some basics** of how QEMU works are understood so there is a clear picture on how difficult is to predict a machine (virtual) behavior (specially when parts of it are being emulated by the host kernel and parts are being emulated by userland binary code).

At the end of this document you will have a basic understanding of most of the intersection points between QEMU and the host OS, and this is where security concerns will rise most of the times (mostly due to memory address space sharing among the parts).

1.a-) QEMU basics

QEMU is a hosted virtual machine emulator that provides a set of different hardware and device models for the guest machine. For the host, QEMU appears as a regular process scheduled by the standard Linux scheduler, with its own process memory. In the process, QEMU allocates a memory region that the guest see as physical, and executes the virtual machine's CPU instructions.

To perform I/O on bare metal hardware, like storage or networking, the CPU has to interact with physical devices performing special instructions and accessing particular memory regions, such as the ones that the device is mapped to.

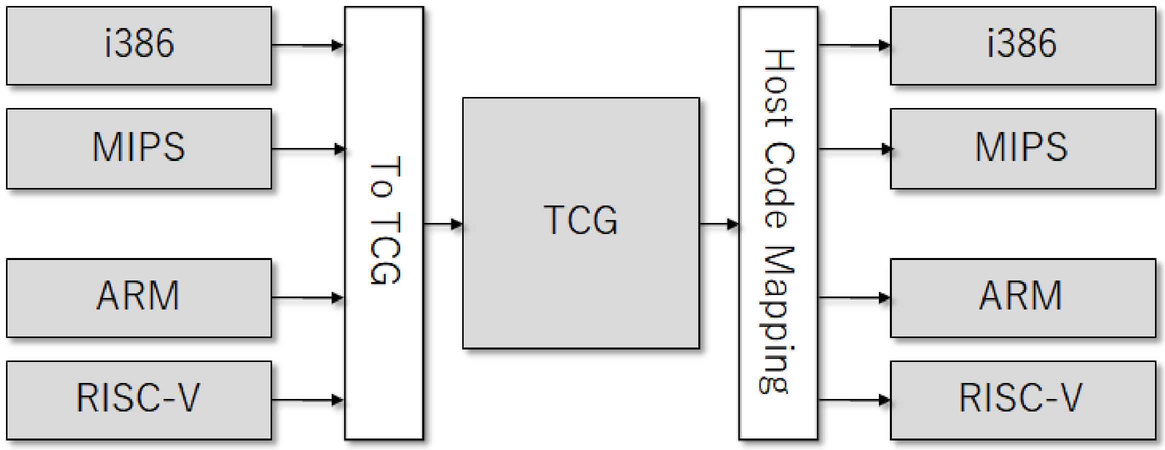
When the guests access these memory regions, control is returned to QEMU, which performs the device's emulation in a transparent manner for the guest.

QEMU has multiple operating modes

- **User-mode emulation**: In this mode QEMU runs single Linux programs that were compiled for a different instruction set. System calls are thunked for endianness and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.
- **System emulation**: In this mode QEMU emulates a full computer system, including peripherals. It can be used to provide virtual hosting of several virtual computers on a single computer. QEMU can boot many guest operating systems and it supports emulating several instruction sets, including x86, MIPS, 32-bit ARMv7, ARMv8, PowerPC, SPARC, ETRAX CRIS and MicroBlaze.

When used as user-mode or system emulator, QEMU can run OSes and programs made for one machine on a different machine. The Tiny Code Generator (TCG) is the core binary translation engine that is responsible for QEMU ability to emulate foreign processors on any given supported host:

The TCG works by translating each guest instruction into a sequence of host instructions. As a result there will be a level of inefficiency which means TCG code will not be as fast as running native code. However with a reasonable host system you can get a pretty good experience, especially when emulating older and slower chips.



The Tiny Code Generator (TCG) aims to remove the shortcoming of relying on a particular compiler, instead incorporating the compiler (code generator) into other tasks performed by QEMU at run time. The whole translation task thus consists of two parts: basic blocks of target code (TBs) being rewritten in TCG ops - a kind of machine-independent intermediate notation - and subsequently this notation being compiled for the host's architecture by TCG (with optional optimisation passes done by the just-in-time compiler (JIR).

KVM Hosting: Here QEMU deals with the setting up and migration of KVM images. It is still involved in the emulation of hardware, but the execution of the guest is done by KVM as requested by QEMU. The picture above showing TCG and the JIT technique does not apply in this case, as KVM only works when HOST and GUEST are in the same architecture (guest binaries are executed without any type of modification/interpretation).

Xen Hosting:: QEMU is involved only in the emulation of hardware; the execution of the guest is done within Xen and is totally hidden from QEMU.

When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the KVM kernel module or Xen hypervisor.

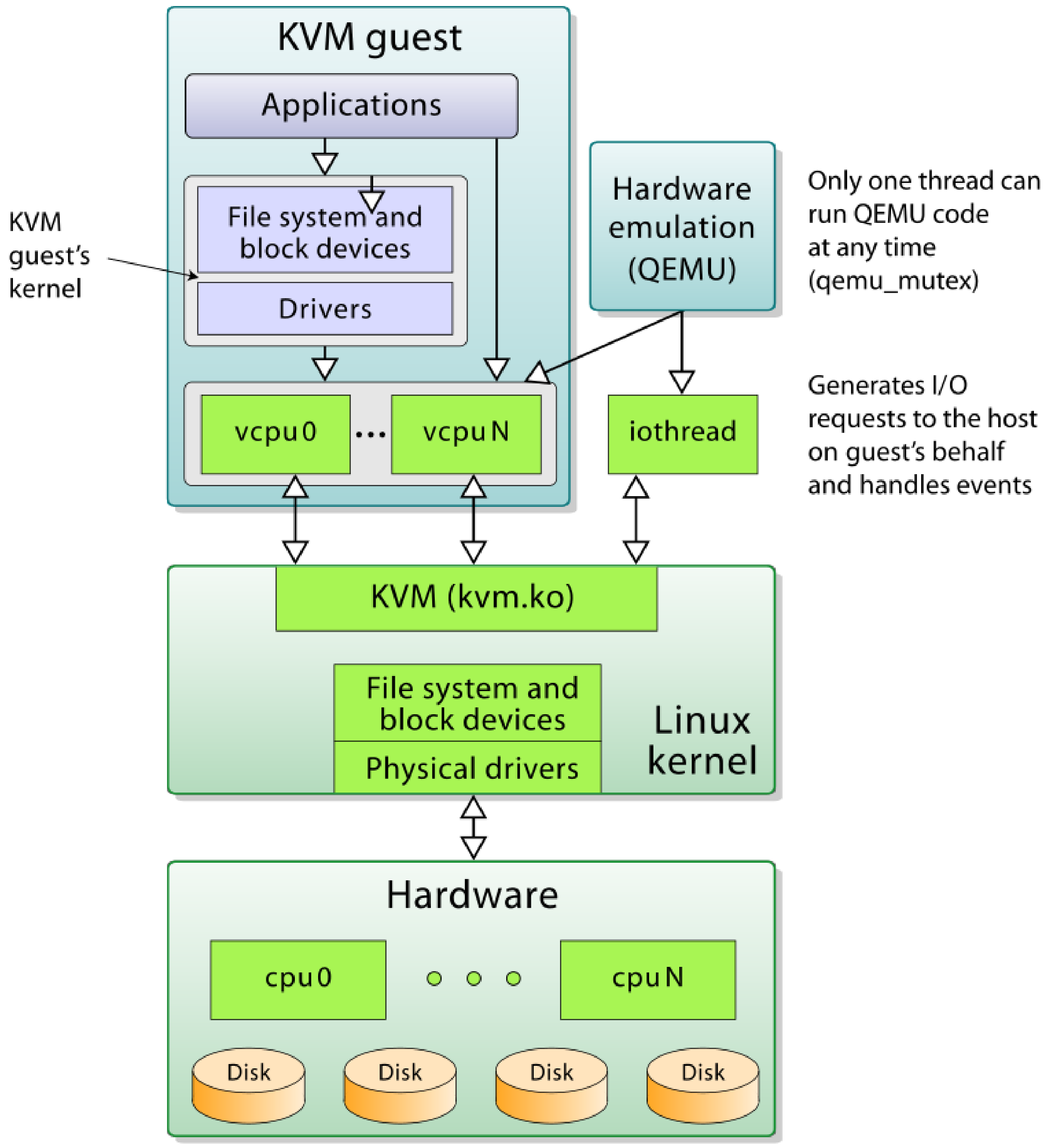
1.b-) KVM basics

Kernel-based Virtual Machine (KVM) is an open source virtualization technology built into Linux. It provides hardware assist to the virtualization software, using built-in CPU virtualization technology to reduce virtualization overheads (cache, I/O, memory) and improving security.

With KVM, QEMU can just create a virtual machine with virtual CPUs (vCPUs) that the processor is aware of, that runs native-speed instructions. When a special instruction is reached by KVM, like the ones that interacts with the devices or to special memory regions, vCPU pauses and informs QEMU of the cause of pause, allowing hypervisor to react to that event.

In the regular KVM operation, the hypervisor opens the device /dev/KVM, and communicates with it using ioctl calls to create the VM, add CPUs, add memory (allocated by QEMU, but physical from the virtual machine's point of view), send CPU interrupts (as an external device would send), etc. For example, one of these ioctl runs the actual KVM vCPU,, blocking QEMU and making the vCPU run until it found an instruction that needs hardware assistance. In that moment, the ioctl returns (this is called vmexit) and QEMU knows the cause of that exit (for example, the offending instruction).

For special memory regions, KVM follows a similar approach, marking memory regions as Read Only or not mapping them at all, causing a vmexit with the KVM_EXIT_MMIO reason.

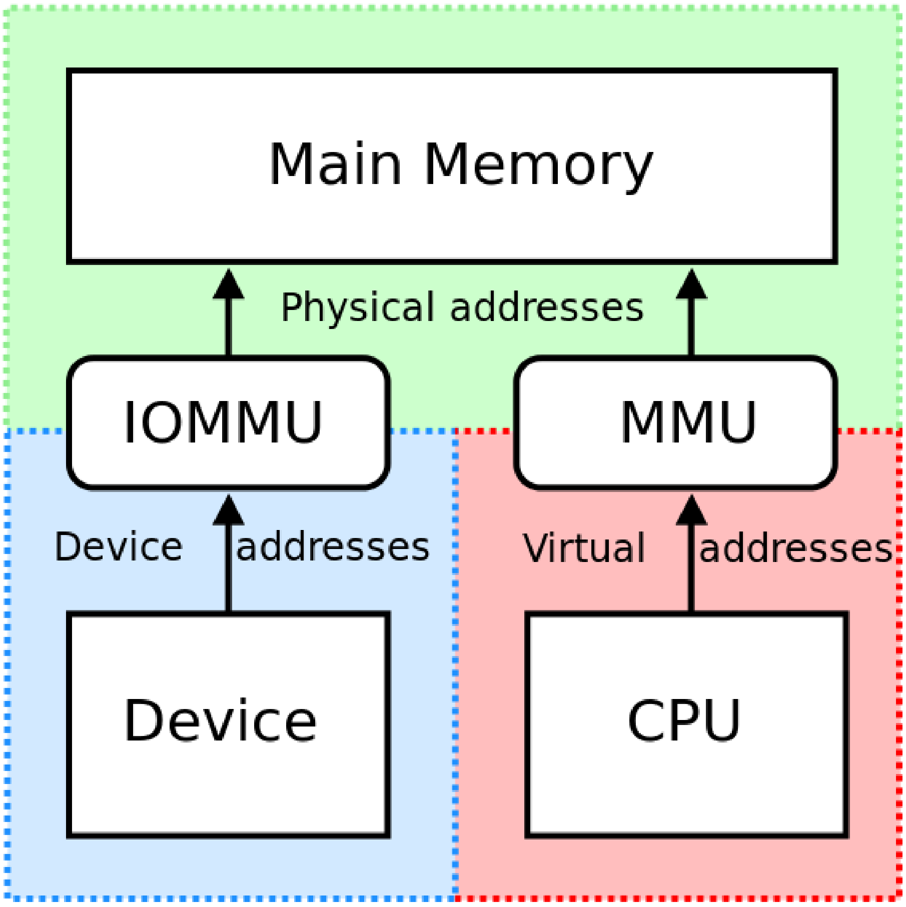

KVM guest
Applications
File system and block devices
Drivers
KVM guest's kernel
Hardware emulation (QEMU)
Only one thread can run QEMU code at any time (qemu_mutex)
vcpu0
...
vcpuN
iothread
Generates I/O requests to the host on guest's behalf and handles events
KVM (kvm.ko)
File system and block devices
Physical drivers
Linux kernel
Hardware
cpu0
cpuN
Disk
Disk
Disk
Disk

1.c-) Device Assignment Methods

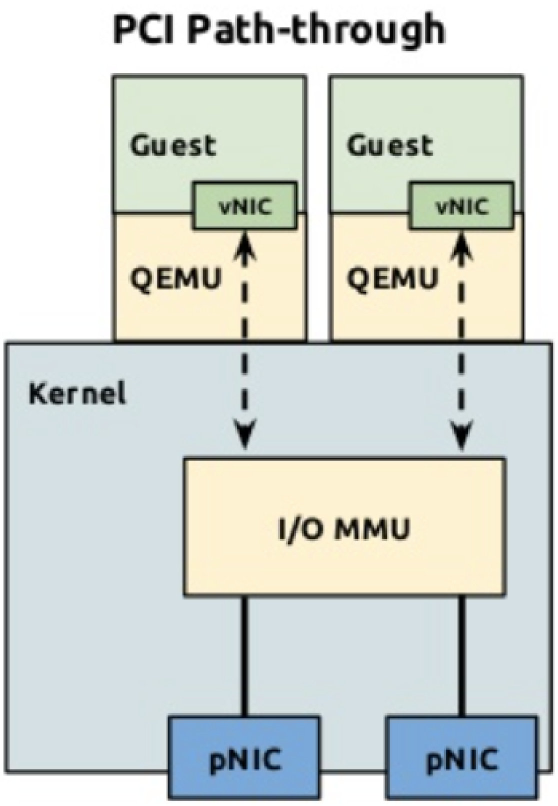
1.c.1-) IOMMU

(its security importance)

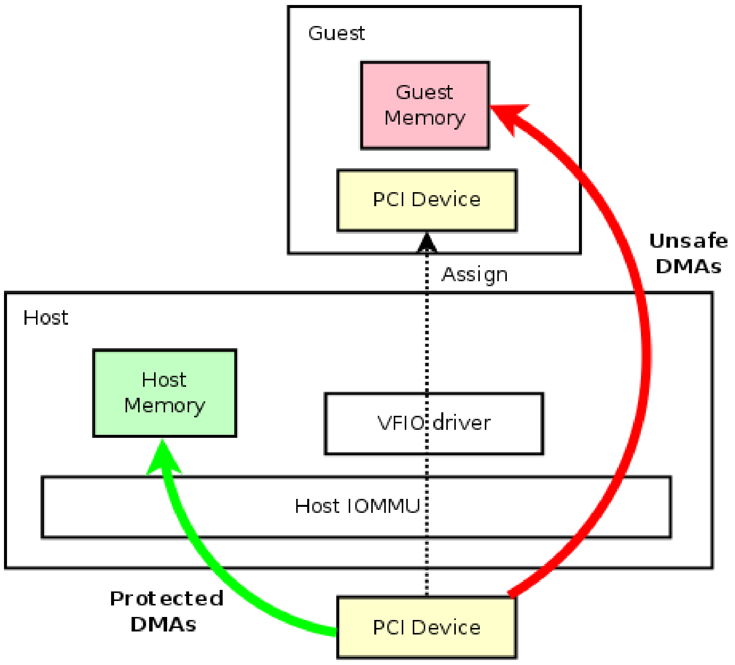
In computing, an input–output memory management unit (IOMMU) is a memory management unit (MMU) that connects a direct-memory-access–capable (DMA-capable) I/O bus to the main memory. Like a traditional MMU, which translates CPU-visible virtual addresses to physical addresses, the IOMMU maps device-visible virtual addresses (also called device addresses or I/O addresses in this context) to physical addresses. Some units also provide memory protection from faulty or malicious devices.



1.c.2-) VFIO



Let's consider a generic PCI device above, which is a real hardware attached to host system. The host can use generic kernel drivers to drive the device. In that case, all the reads/writes of that device will be protected by host IOMMU (part of host chipset), which is safe. The protected DMAs are shown in green arrow.



The PCI device can also be assigned to a guest. By leveraging VFIO driver in the host kernel, the device can be exclusively managed by any userspace programs, like QEMU. In the guest with assigned device, we should be able to see exactly the same device just like in the host. Here, the hypervisor is capable of modifying the device information, like capability bits, etc... but the control is up to the userspace program.

When the device is assigned to the guest, guest memory address space is totally exposed to the hardware PCI device. So there would have no protection when the device do DMAs to the guest system, especially writes. Malicious writes can corrupt the guest in no time. Those unsafe DMAs are shown with a red arrow.

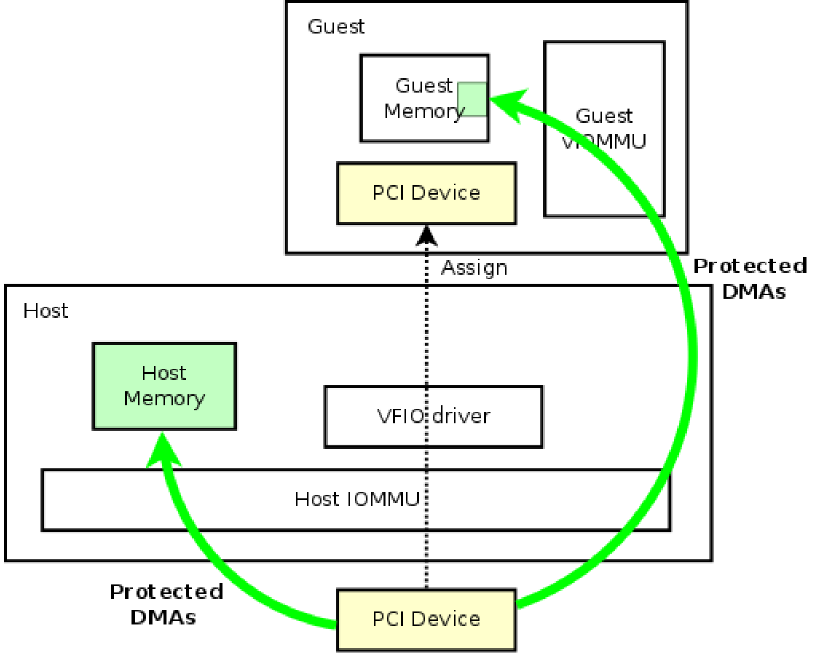
That is why vIOMMU is needed in the guest for security reasons

1.c.3-) vIOMMU

To protect the guest memory from malicious assigned devices, QEMU might give vIOMMU emulation to the guest, just like what the host IOMMU does to the host.

QEMU vIOMMU tries to emulate the effect of a REAL IOMMU (part of the CPU architecture). It emulates all registers, translations and interrupts re-mappings. Mechanism is similar to emulating other devices: QEMU constructs the ACPI DMAR table so guest OS thinks there is an IOMMU available. By telling KVM to mark (the vCPU register space) as "not present", each time QEMU guest (accelerated or not) tries to access IOMMU it gets trapped and access is handled through trapped faults.

Then the picture will be like:



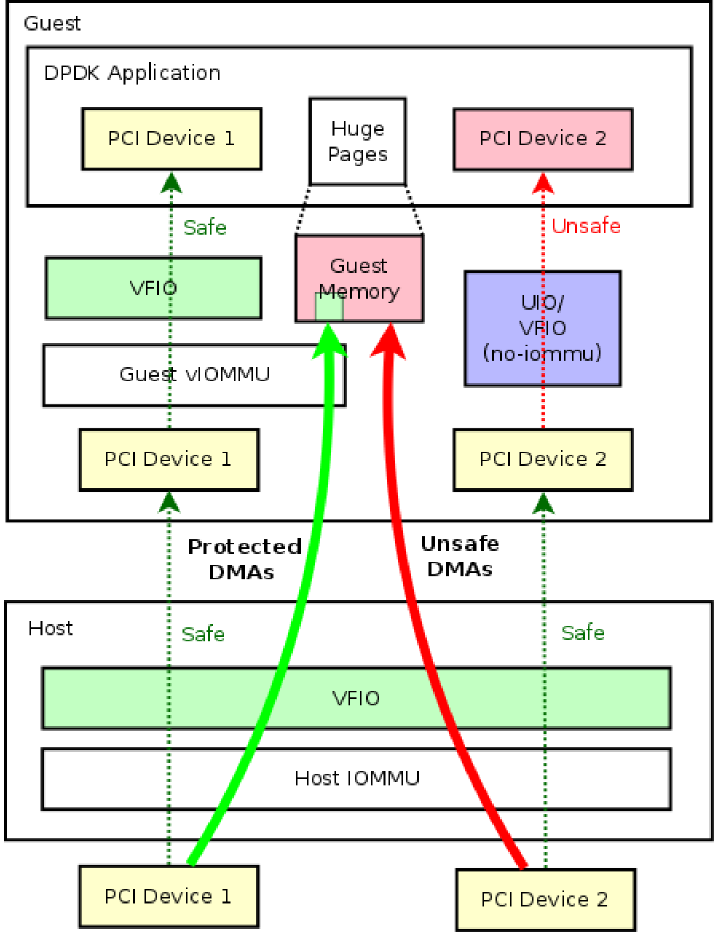
In the above figure, the only difference from previous case is that we introduced guest vIOMMU to do DMA protections. With that, guest DMAs are safe now (if we consider miss-behaving attached - to guests - devices).

1.c.4-) vIOMMU and DPDK

DPDK (the so-called DataPlane Development Kit) is vastly used in high performance scenarios, which moved the kernel space drivers into userspace for the sake of even better performance. Normally, the DPDK program can run directly inside a bare metal to achieve the best performance with specific hardware. Meanwhile, it can also be run inside guest to drive either an assigned device from host, or an emulated device like VIRTIO ones. For the guest DPDK use case mentioned, host can still continue leveraging DPDK to maximum the packet delivery in the virtual switches. OVS-DPDK is a good example.

Nevertheless, DPDK introduced a problem that since we cannot really trust any userspace application program, we cannot trust DPDK applications as well, especially if it can have full access to the system memory via the hardware and taint the kernel address space. Here vIOMMU protects not only the malicious devices like hardware errors, it also protects guest from buggy userspace drivers like DPDK (via VFIO driver in the guest).

Actually there are at least two ways that DPDK applications can manage a device in the userspace (and these methods are mostly general as well not limited to DPDK use cases): with and without a vIOMMU.

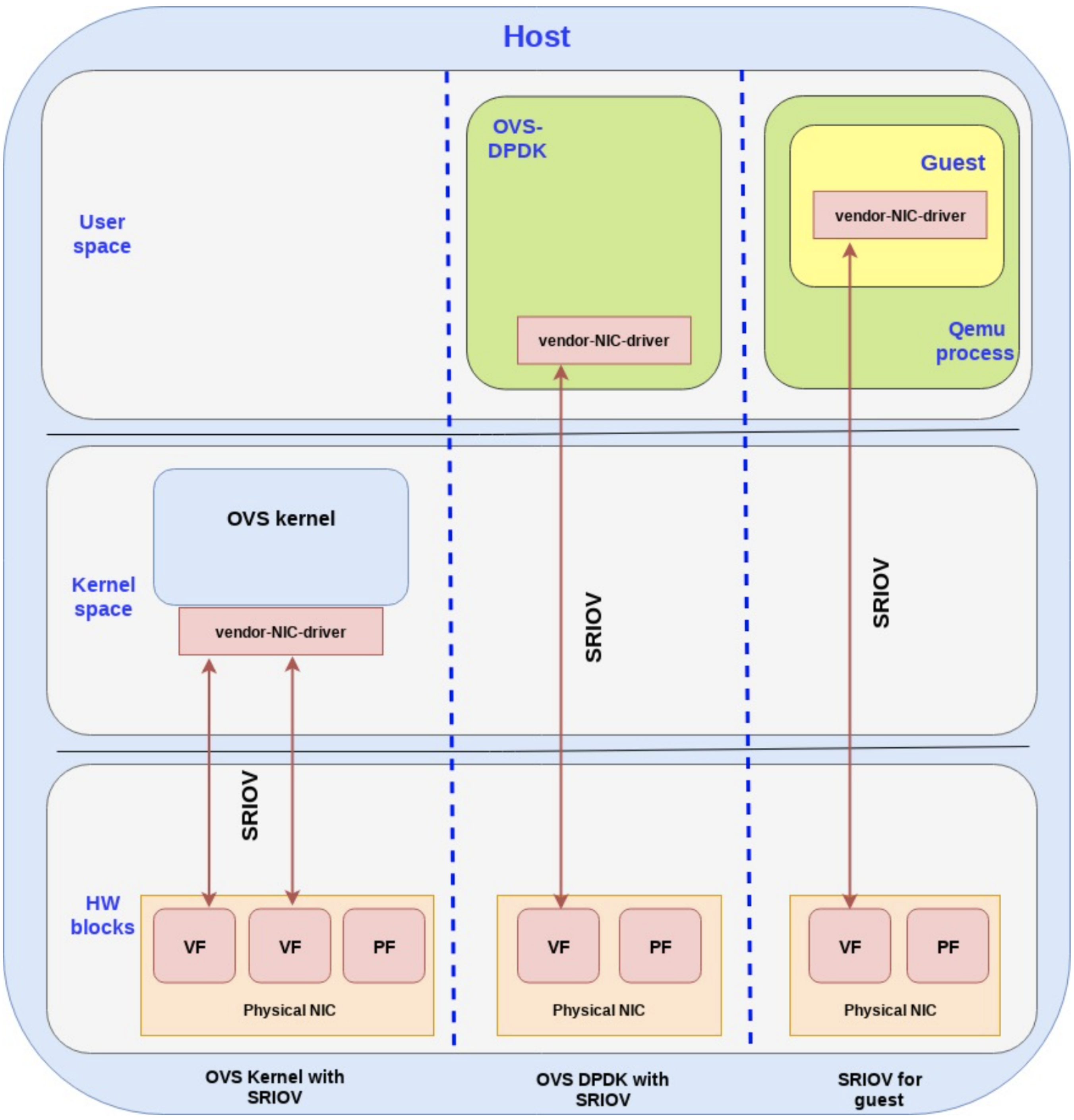


In above case, PCI Device 1 and PCI Device 2 are two devices that are assigned to guest DPDK applications. In the host, both of the devices are assigned to guest using kernel VFIO driver. While in the guest, when we assign devices to DPDK applications, we can use one of the two methods mentioned above. However, only if we assign device with generic VFIO driver (which requires a vIOMMU) we get a safely assigned device. Assigning the device by "VFIO no-iommu mode" is unsafe.

1.c.5-) SR-IOV

The Direct Assignment method of virtualization provides very fast I/O. However, it prevents the sharing of I/O devices. SR-IOV provides a mechanism by which a Single Root Function (for example a single Ethernet Port) can appear to be multiple separate physical devices.

An image illustrating SR-IOV usage:



SR-IOV enables a Single Root Function (for example, a single Ethernet port), to appear as multiple, separate, physical devices. A physical device with SR-IOV capabilities can be configured to appear in the PCI configuration space as multiple functions. Each device has its own configuration space complete with Base Address Registers (BARs).

SR-IOV uses two PCI functions:

1. Physical Functions (PFs) are full PCIe devices that include the SR-IOV capabilities. Physical Functions are discovered, managed, and configured as normal PCI devices. Physical Functions configure and manage the SR-IOV functionality by assigning Virtual Functions.

2. Virtual Functions (VFs) are simple PCIe functions that only process I/O. Each Virtual Function is derived from a Physical Function. The number of Virtual Functions a device may have is limited by the device hardware. A single Ethernet port, the Physical Device, may map to many Virtual Functions that can be shared to virtual machines.

The hypervisor can assign one or more Virtual Functions to a virtual machine, an OS process or its own kernel. If assigned to a virtual machine, the virtual function's configuration space is assigned to the configuration space presented to the guest.

> Each Virtual Function can only be assigned to a single guest at a time, as Virtual Functions require real hardware resources. A virtual machine can have multiple Virtual Functions. A Virtual Function appears as a network card in the same way as a normal network card would appear to an operating system.

The SR-IOV drivers are implemented in the kernel. The core implementation is contained in the PCI subsystem, but there must also be driver support for both the Physical Function (PF) and Virtual Function (VF) devices. An SR-IOV capable device can allocate VFs from a PF. The VFs appear as PCI devices which are backed on the physical PCI device by resources such as queues and register sets.

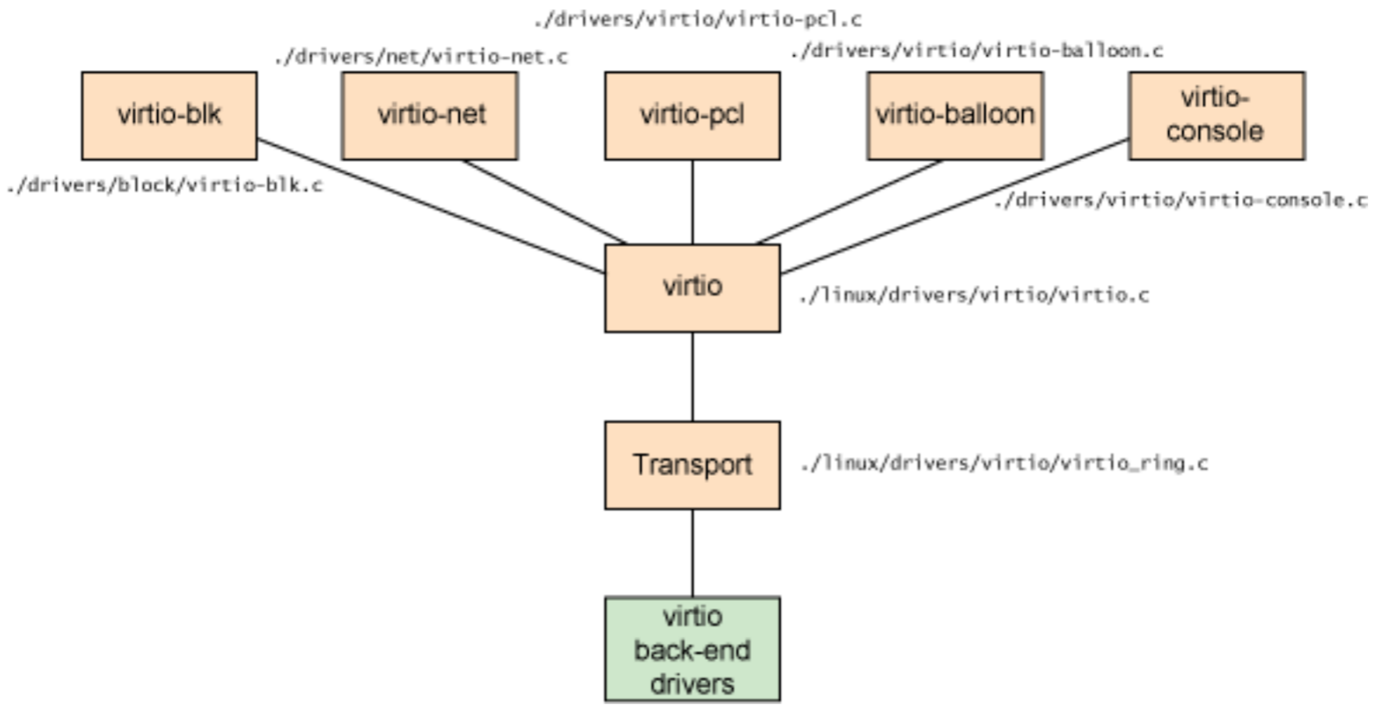
Some advantages for using SR-IOV:

- When an SR-IOV VF is assigned to a virtual machine, it can be configured to (transparently to the virtual machine) place all network traffic leaving the VF onto a particular VLAN. The virtual machine cannot detect that its traffic is being tagged for a VLAN, and will be unable to change or eliminate this tagging.
- Virtual Functions have near-native performance and provide better performance than paravirtualized (virtio) drivers and emulated access. Virtual Functions provide data protection between virtual machines on the same physical server as the data is managed and controlled by the hardware.

> This method is very used in environments with NICs w/ 10GbE or more, supporting virtual functions, and able to do RoCE (like Mellanox ConnectX cards).

1.d-) Device Para-Virtualization

1.d.1-) virtio


./drivers/virtio/virtio-pcl.c
./drivers/net/virtio-net.c
./drivers/virtio/virtio-balloon.c
virtio-blk
virtio-net
virtio-pcl
virtio-balloon
virtio-console
./drivers/block/virtio-blk.c
./drivers/virtio/virtio-console.c
virtio
./linux/drivers/virtio/virtio.c
Transport
./linux/drivers/virtio/virtio_ring.c
virtio back-end drivers

Virtio is an open specification for virtual machines' data I/O communication, offering a straightforward, efficient, standard and extensible mechanism for virtual devices, rather than boutique per-environment or per-OS mechanisms. It uses the fact that the guest can share memory with the host for I/O to implement that.

The VIRTIO specification is based on two elements: devices and drivers. In a typical implementation, the hypervisor exposes the VIRTIO devices to the guest through a number of transport methods. By design they look like physical devices to the guest within the virtual machine.

The most common transport method is PCI or PCIe bus. However, the device can be available at some predefined guest's memory address (MMIO transport). These devices can be completely virtual with no physical counterpart or physical ones exposing a compatible interface.

The typical (and easiest) way to expose a VIRTIO device is through a PCI port since we can leverage the fact that PCI is a mature and well supported protocol in QEMU and Linux drivers. Real PCI hardware exposes its configuration space using a specific physical memory address range (i.e., the driver can read or write the device's registers by accessing that memory range) and/or special processor instructions. In the VM world, the hypervisor captures accesses to that memory range and performs device emulation, exposing the same memory layout that a real machine would have and offering the same responses. The VIRTIO specification also defines the layout of its PCI Configuration space, so implementing it is straightforward.

When the guest boots and uses the PCI/PCIe auto discovering mechanism, the virtio devices identify themselves with with the PCI vendor ID and their PCI Device ID. The guest's kernel uses these identifiers to know which driver must handle the device. In particular, the linux kernel already includes virtio drivers.

The VIRTIO drivers must be able to allocate memory regions that both the hypervisor and the devices can access for reading and writing, i.e., via memory sharing. We call data plane the part of the data communication that uses these memory regions, and control plane the process of setting them up:

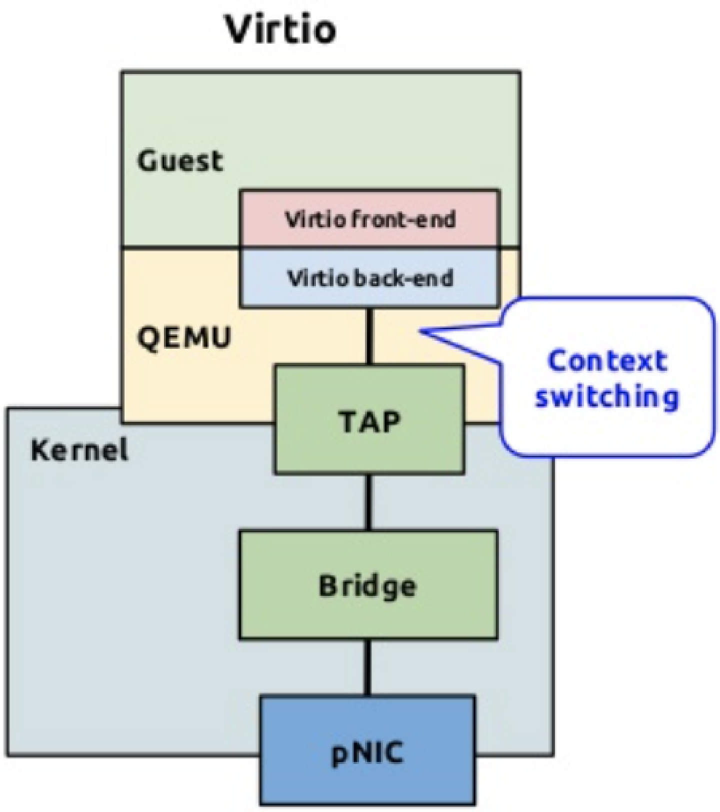
- VIRTIO element (1): **control plane** used for capability exchange negotiation between the host and guest both for establishing and terminating the data plane.
- VIRTIO element (2): **data plane**: used for transferring the actual data (packets) between host and guest.

We can also split VIRTIO into those different parts:

- VIRTIO part (1): **virtio spec** defines how to create a control plane (1) and the data plane (2) between the guest and host. The data plane (2) is composed of buffers and rings layouts defined in the spec.

> The control plane (1) for VIRTIO is implemented in the QEMU process based on the VIRTIO spec however the data plane (2) is not. Thus the question is why wasn't the data plane (2) also implemented in the QEMU process ? The answer is performance.

> If VIRTIO spec data plane was implemented in QEMU, for every packget going from kernel to the guest, and vice-versa, theere would be a context switch. Context switches are expensive adding latency to the application and requires more processing time (as QEMU is yet another linux process).



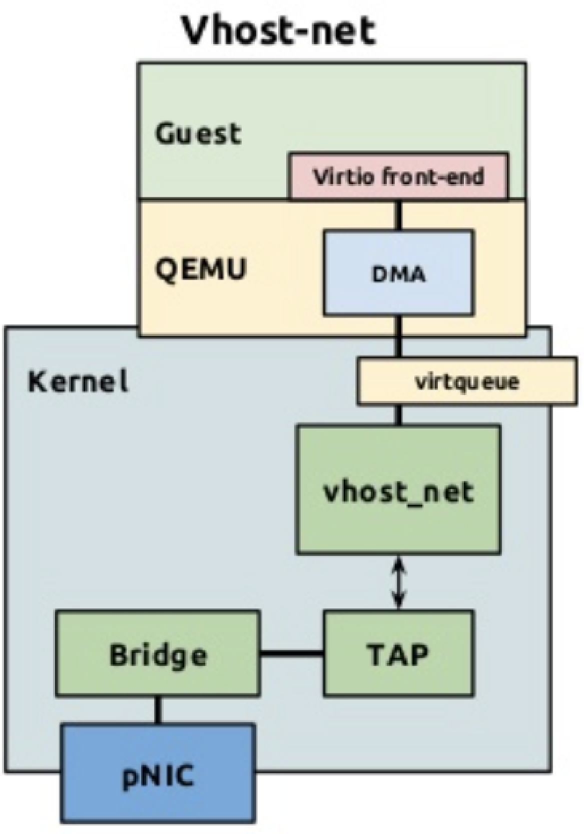
- VIRTIO part (2): **vhost protocol** allows the VIRTIO dataplane implementation to be offloaded to another element (**user process or kernel module**) in order to enhance performance.

> Vhost implements a data plane going directly from the host kernel to the guest, bypassing QEMU process entirely. The vhost only describes how to establish the data plane, however. Whoever implements it is also expected to implement the ring layout for describing the data buffers (both host and guest) and the actual send/receive packets.

> The vhost protocol can be implemented in the kernel (vhost-net) or in the user space (vhost-user). The vhost-net/virtio-net architecture described in this post focuses on the kernel implementation also known as vhost-net.

1.d.2-) virtio-net <=> vhost-net

- vhost-net = backend component: host side of the VIRTIO interface
- virtio-net = frontend component: guest side of the VIRTIO interface



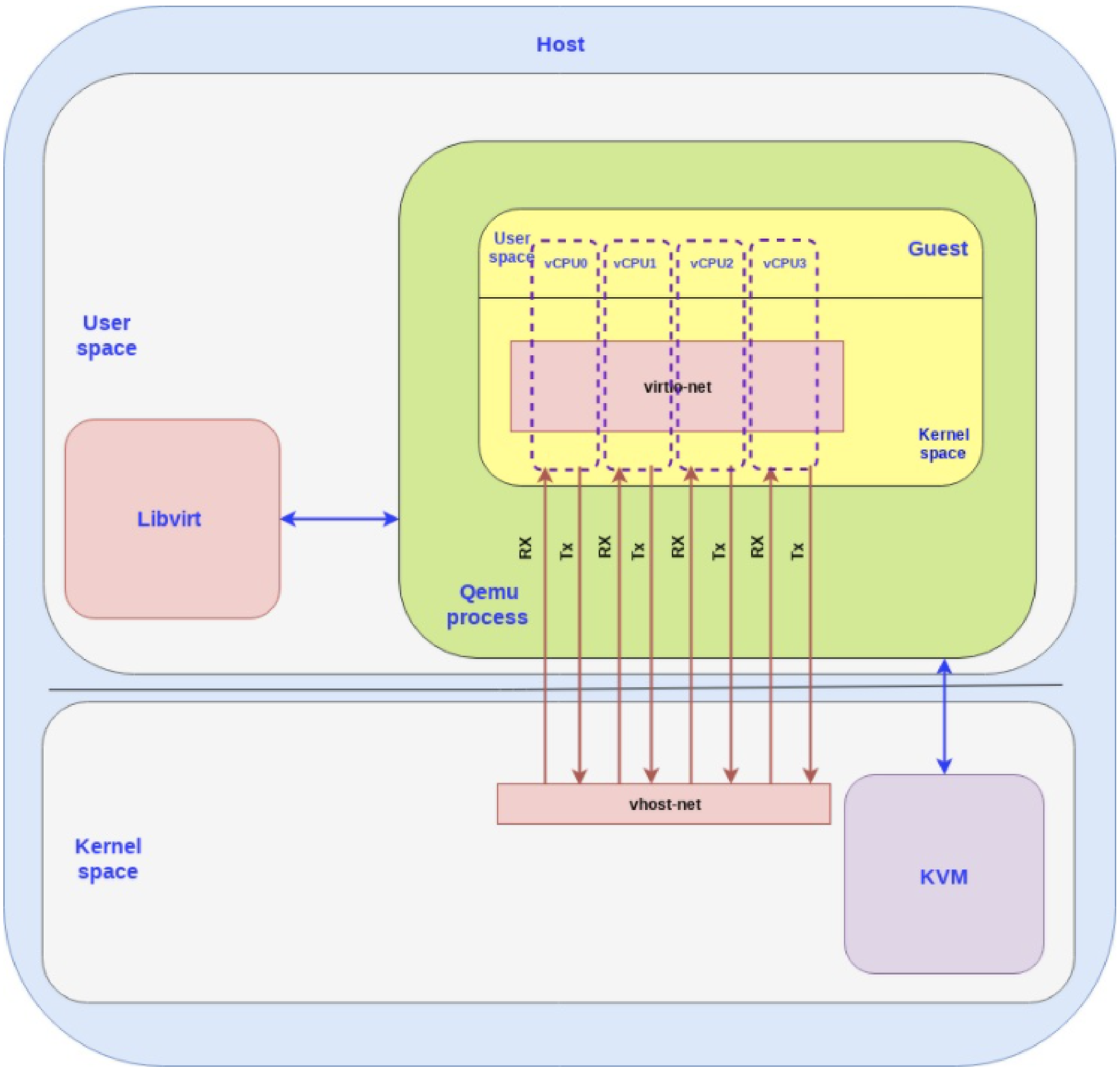
> vhost-net is part of the host kernel but and yet still called 'driver'

Both components:

- vhost-net (backend)
- virtio-net (frontend)

Have separate control plane (1) and data plane (2) between the backend and frontend:

- control plane implements VIRTIO spec for vhost-net (module) <-> QEMU-process communication.
- vhost protocol sets communication for data plane to fwd pkts from host to guest using shared memory.
- data plane communication is accomplished through dedicated queues
- each guest vCPU has at least 1 RX/TX queue



Up to this point we have described how the guest can pass the packets to the host kernel using the virtio-networking interface.

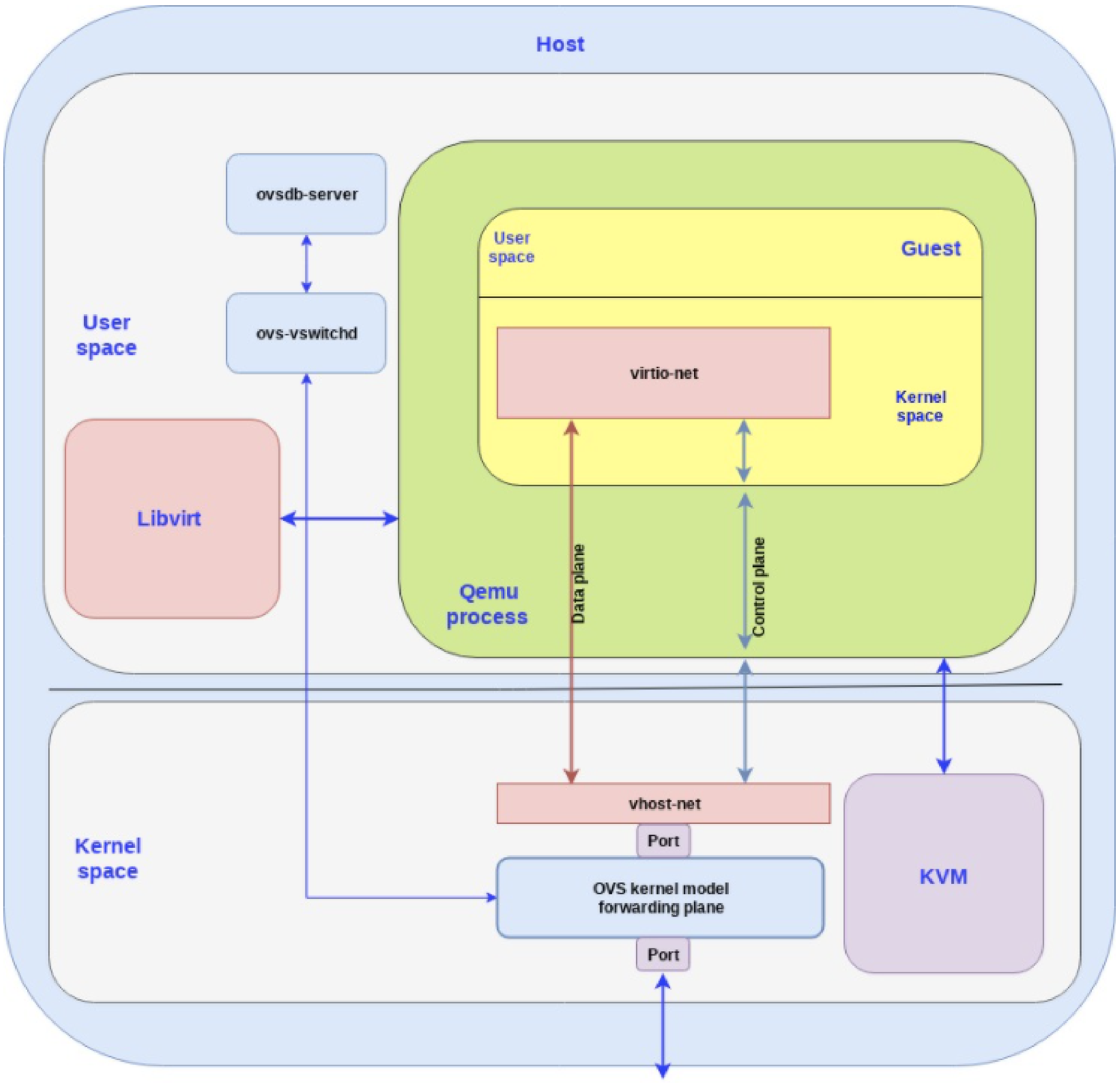
> ATTENTION: This is the method that will be used as an example further in this document.

1.d.3-) virtio-net <=> openvswitch

In order to forward these packets to other guest running on the same host, or outside the host, one may use openvswitch. OVS is a software switch which enables the packet forwarding inside the kernel. It's composed of a userspace part and a kernel part:

- User space - database (ovsdb-server) and an OVS daemon/controller (ovs-vswitchd)
- Kernel space - ovs kernel (module responsible for the datapath or forwarding plane)

The OVS controller communicates both with the database server and the kernel forwarding plane. To push packets in and out of the OVS we use Linux ports.



In the example above we have one port that connects the OVS kernel forwarding plane to a physical NIC while the other port connects to the vhost-net backend (which will communicate with the virtio-net frontend).

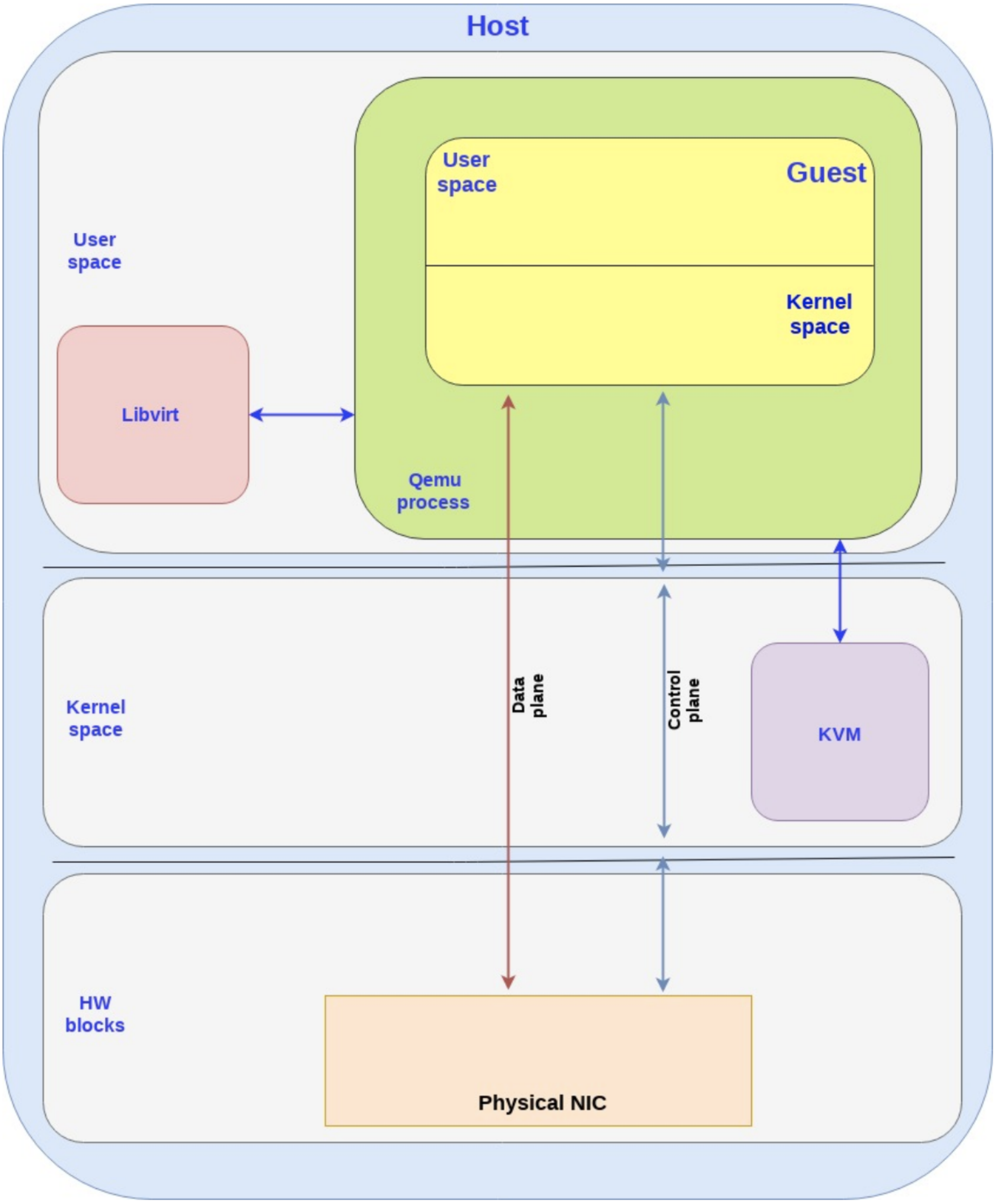
1.d.4-) virtio-net <=> SR-IOV

Single root I/O virtualization (SR-IOV) is a standard for a type of PCI device assignment that can share a single device to multiple virtual machines. In other words, it allows different VMs in a virtual environment to share a single NIC. This means we can have a single root function such as an Ethernet port appear as multiple separated physical devices which address our problem of creating "virtual ports" in the NIC.

You can think of SR-IOV as a virtio-net (front-end) vhost-net (backend) but, instead of having the vhost protocol in between QEMU virtual machine driver and the host kernel, with the vhost-net part implemented directly by a virtual function of the hardware itself.

Similar to previous VIRTIO architectures, we separate our communication channel to data plane and control plane towards the NIC:

- Control plane: config changes and capability negotiation between the NIC and guest (establishing and terminating the data plane)
- Data plane: transfer the actual data (packets) between NIC and guest. When connecting the NIC directly to the guest, this implies that the NIC is required to support the VIRTIO ring layout.



> This is the model we will be analyzing further in a virtual machine XML example.

- The data plane goes directly from the NIC to the guest and this is implemented by a shared memory the guest and the NIC can access (shared by the guest), bypassing the host kernel (differently than vhost-net when the host kernel is the one accessing the shared memory).

> both sides need to use the exact same ring layout or translations will be required and translations have a performance penalty attached.

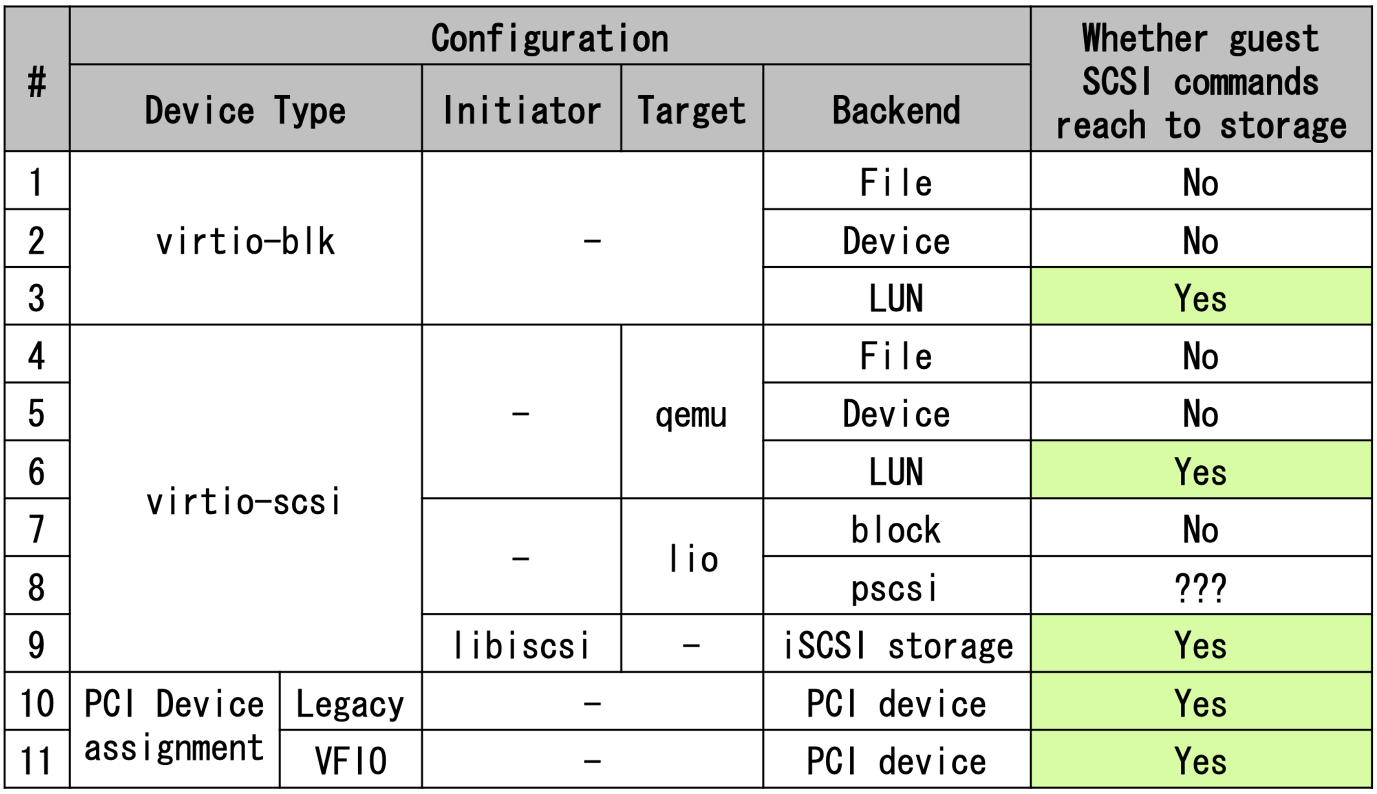
1.e-) Block Devices, LUNs and Disks

(virtio-blk, virtio-scsi, PCI/VFIO assignment)

Being VIRTIO a communication mechanism for para-virtualized devices, it has become the default communication mechanism between the QEMU process (hypervisor) and the virtual machine device drivers (para-virtualized devices). Instead like previous items, explaining how VIRTIO communication works, or how vhost might offload the packet processing work, this item lists most, if not all, possible ways to virtualize disks to the virtual machines.

> From now on this document might address Block Devices, LUNs and or Disks merely as "disks" or "vdisks", depending on the context.

There are different ways to deliver disks to a virtual machine which can either try to maximize throughput, flexibility or resource-sharing. The illustration bellow summarizes the different para-virtualized device types QEMU can present to a guest regarding its vdisks:



#	Configuration					Whether guest SCSI commands reach to storage
	Device Type		Initiator	Target	Backend	
1	virtio-blk		-		File	No
2					Device	No
3					LUN	Yes
4	virtio-scsi		-	qemu	File	No
5					Device	No
6					LUN	Yes
7			-	lio	block	No
8					pscsi	???
9			libiscsi	-	iSCSI storage	Yes
10	PCI Device assignment	Legacy	-		PCI device	Yes
11		VFIO	-		PCI device	Yes

The vdisks functionalities - and their intersection with the Host OS in regards to resource sharing and/or security surface - varies depending on the combination of the device type, being delivered to the virtual machine, and the backend used for this device type (whether its a file, an entire device or a LUN). Depending on the configuration, SCSI commands can either be fully emulated, blocked or passed-by.

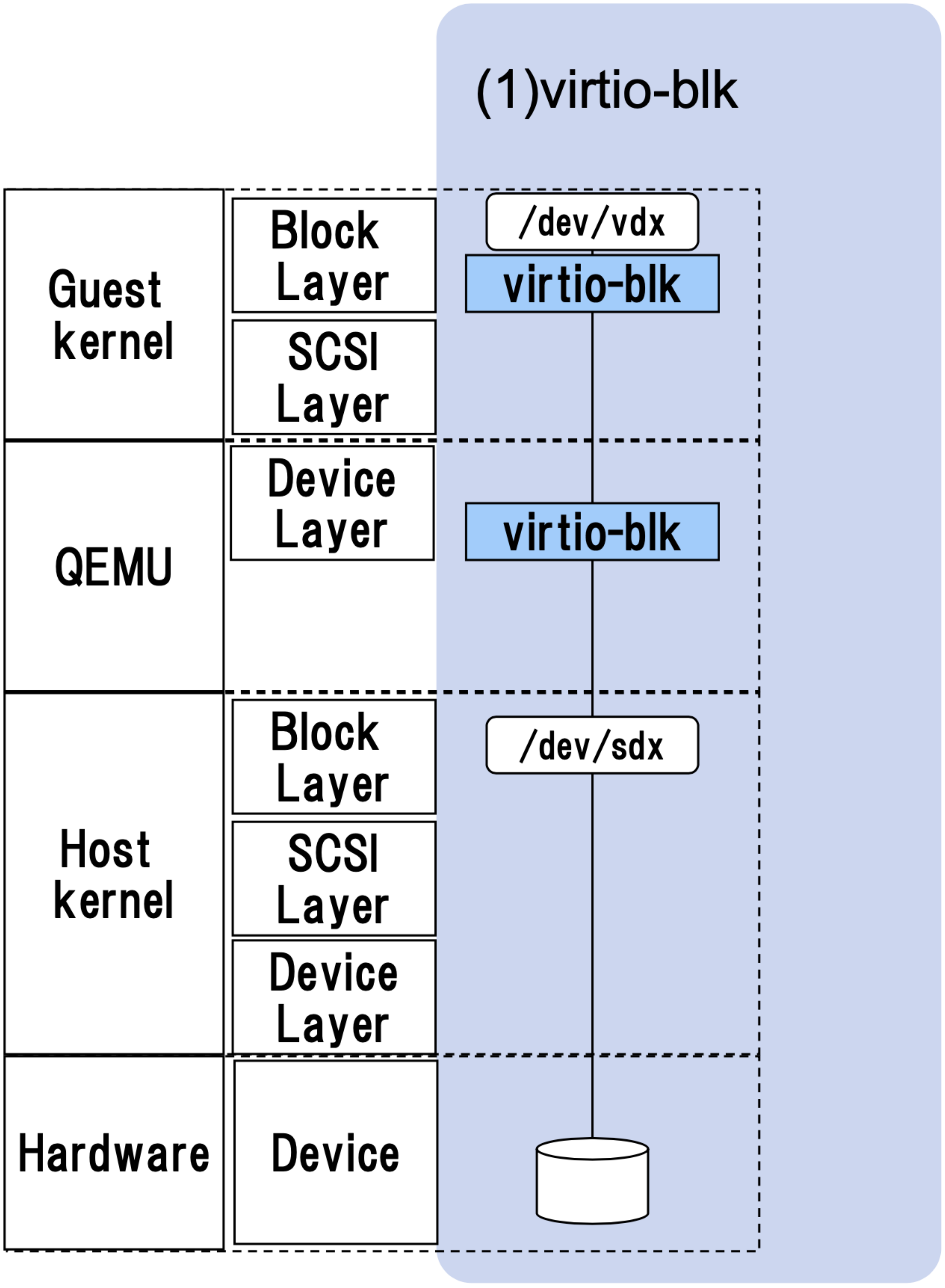
1.e.1-) virtio-blk

The virtio-blk device presents a block device to the virtual machine. Each virtio-blk device appears as a disk inside the guest. virtio-blk was available before virtio-scsi and is the most widely deployed VIRTIO storage controller. It offers high performance thanks to a thin software stack and is therefore a good choice when performance is a priority.

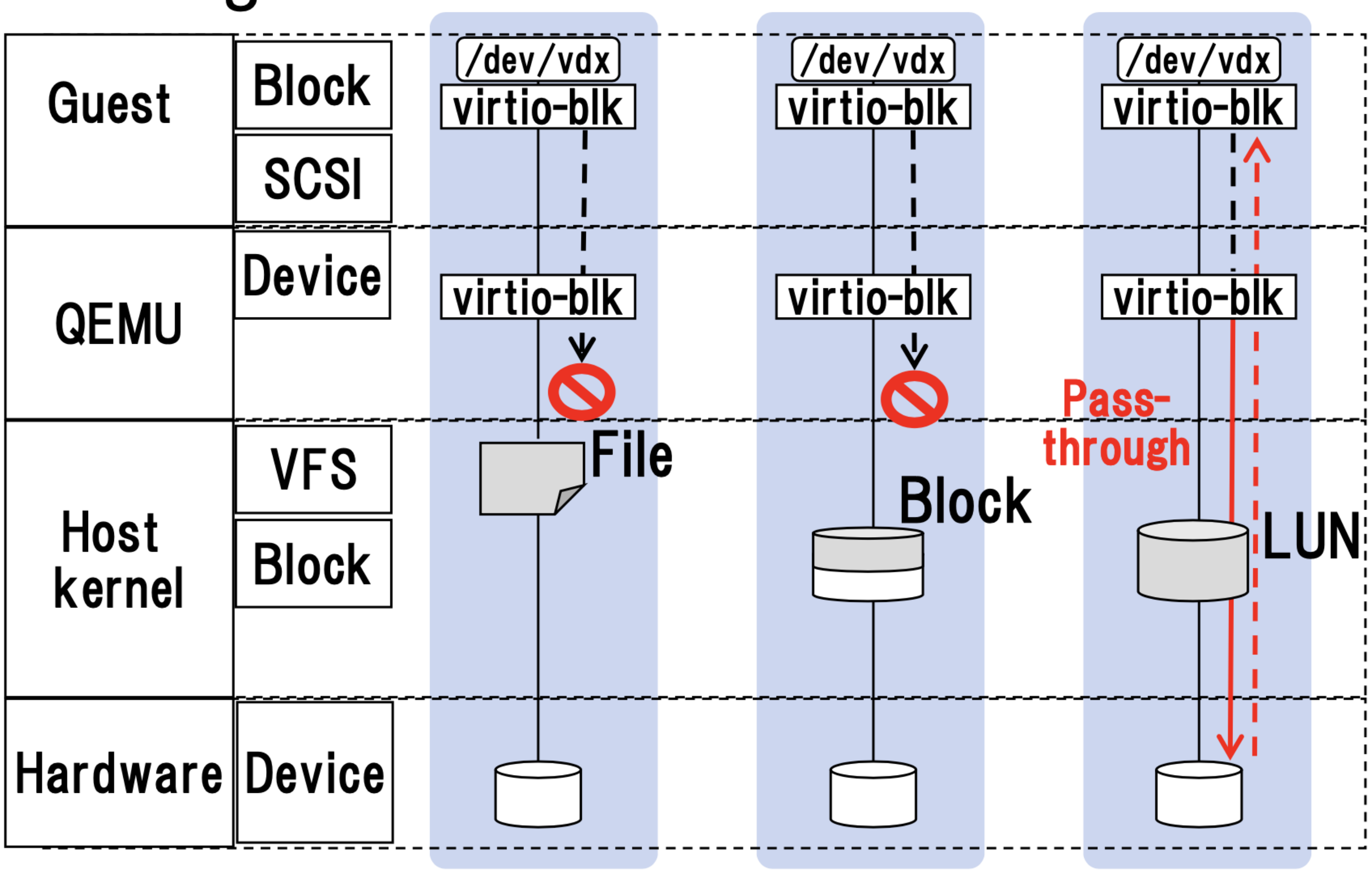
> any application that sends SCSI commands are better served by the virtio-scsi device, which has full SCSI support

Virtual machines that require access to many disks can hit limits based on availability of PCI slots, which are under contention with other devices exposed to the guest, such as NICs. For example a typical i440fx machine type default configuration allows for about 28 disks. It is possible to use multi-function devices to pack multiple virtio-blk devices into a single PCI slot at the cost of losing hotplug support, or additional PCI busses can be defined.

> sometimes it is simpler to use a single virtio-scsi PCI adapter instead.



- virtio-blk SCSI handling



> SCSI passthrough was removed from the Linux virtio-blk driver in v5.6 in favor of using virtio-scsi

1.e.2-) virtio-scsi

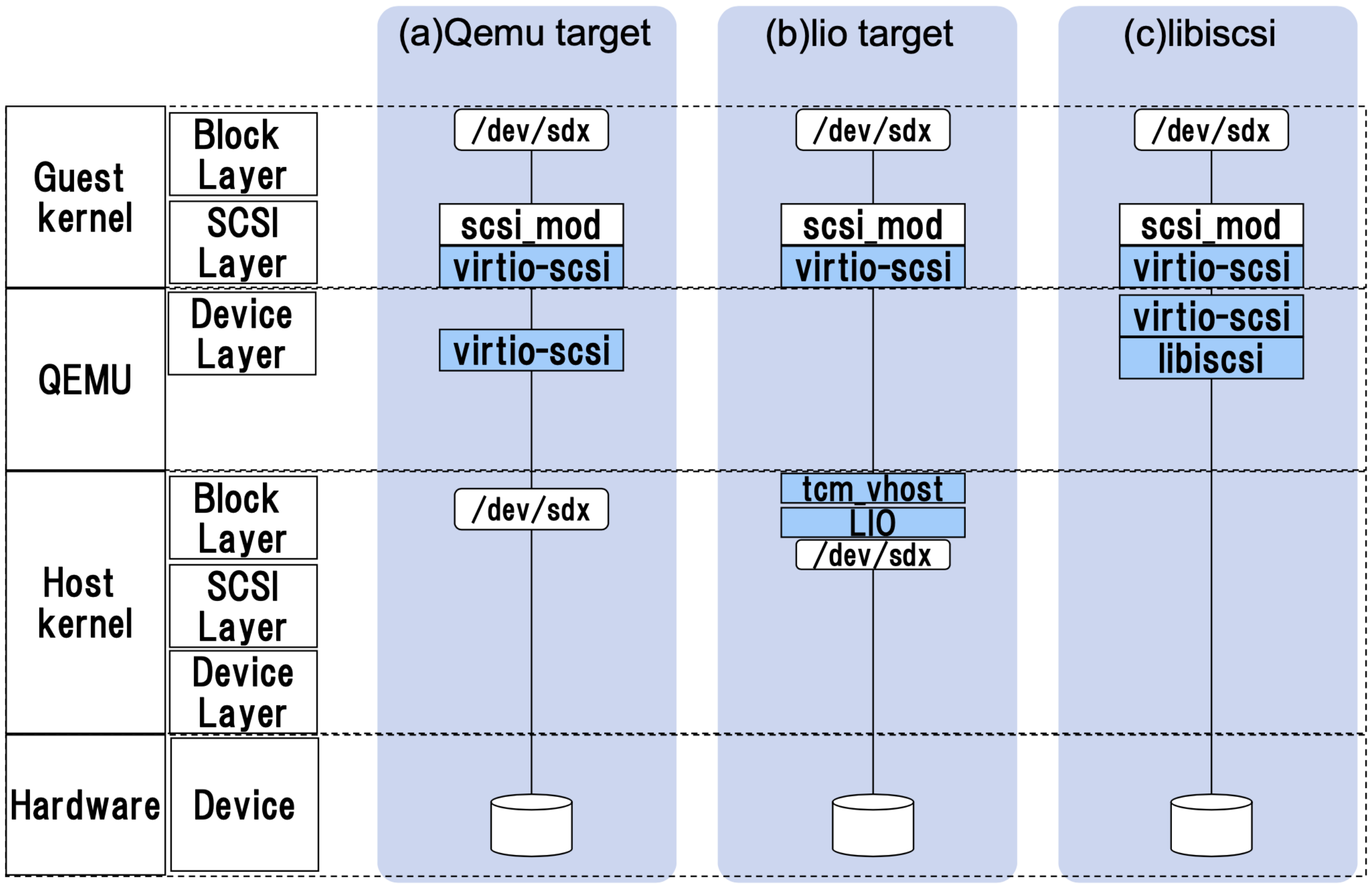
The virtio-scsi device presents a SCSI Host Bus Adapter to the virtual machine. SCSI offers a richer command set than virtio-blk and supports more use cases.

Each device supports up to 16,383 LUNs (disks) per target and up to 255 targets. This allows a single virtio-scsi device to handle all disks in a virtual machine, although defining more virtio-scsi devices makes it possible to tune for NUMA topology.

Emulated LUNs can be exposed as hard disk drives or CD-ROMs. Physical SCSI devices can be passed through into the virtual machine, including CD-ROM drives, tapes, and other devices besides hard disk drives.

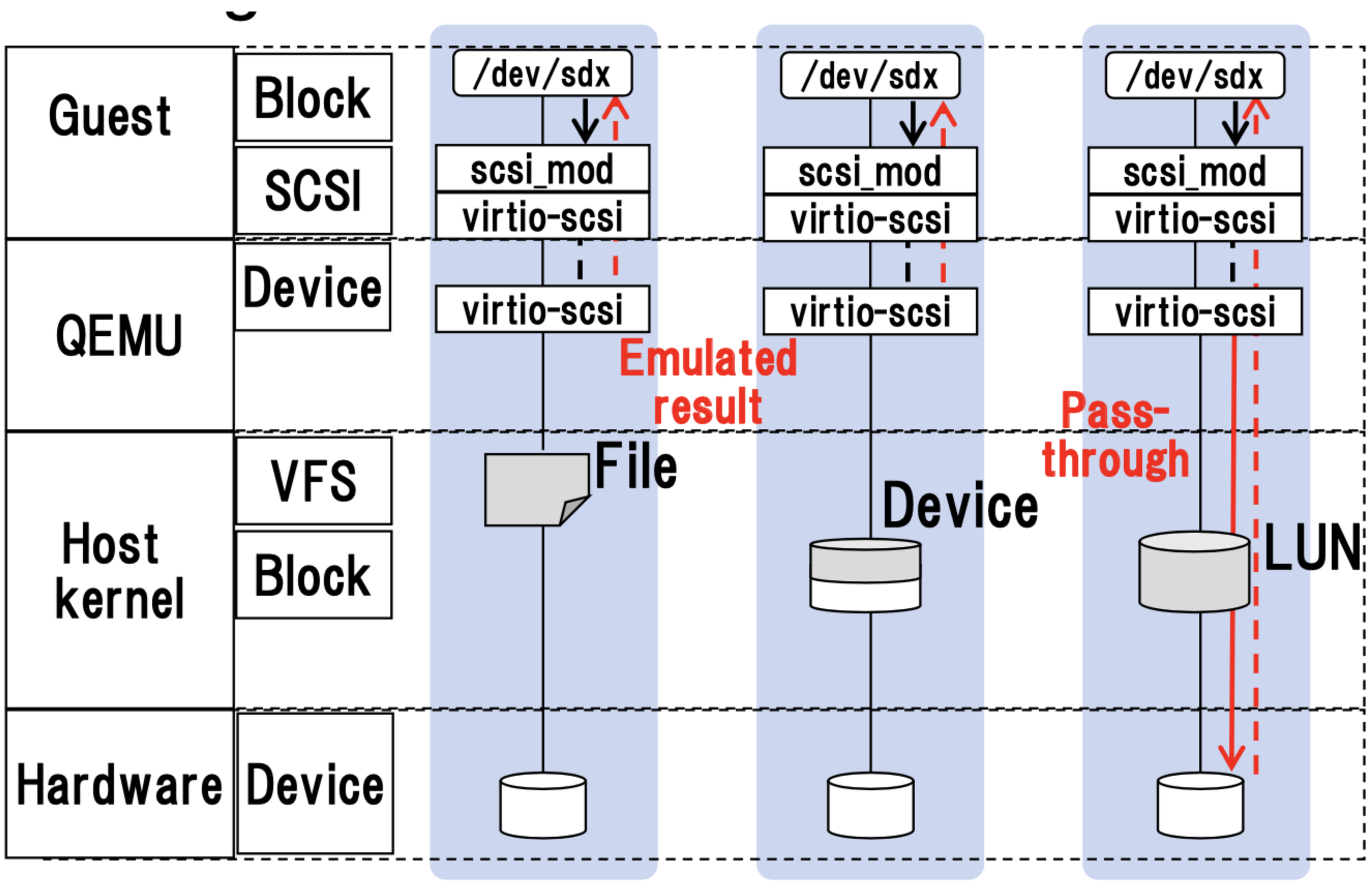
> Clustering software uses SCSI Persistent Reservations and is usually only supported by virtio-scsi, not by virtio-blk.

There are different backends to support a virtio-scsi device being given to a virtual machine:



i. virtio-scsi: QEMU target SCSI handling

QEMU process has different threads, being one of them responsible for the guest's vdisk I/O. Instead of emulation (which would cause the vCPU to exit HW acceleration) the guest communicates with QEMU using the VIRTIO protocol and QEMU does the I/O to the backing device. (files: qcow2, raw ; devices or LUNs).



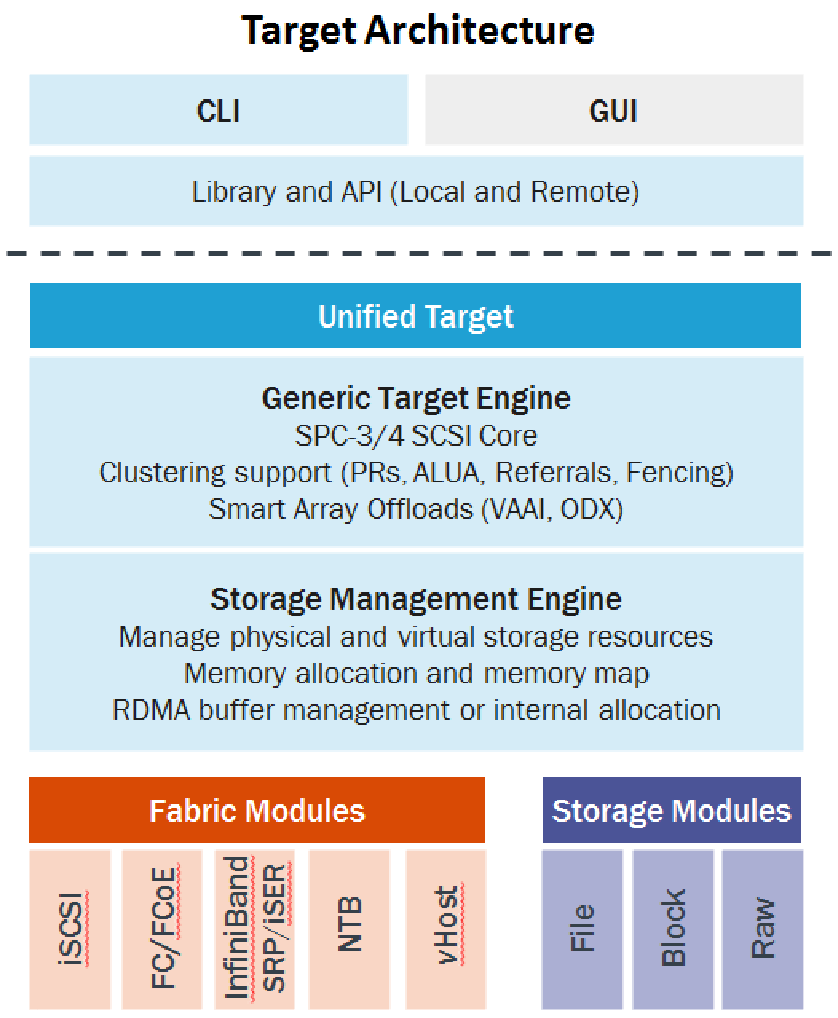
ii. virtio-scsi: LIO target SCSI handling

First, let's see what Linux I/O (LIO) project stands for:

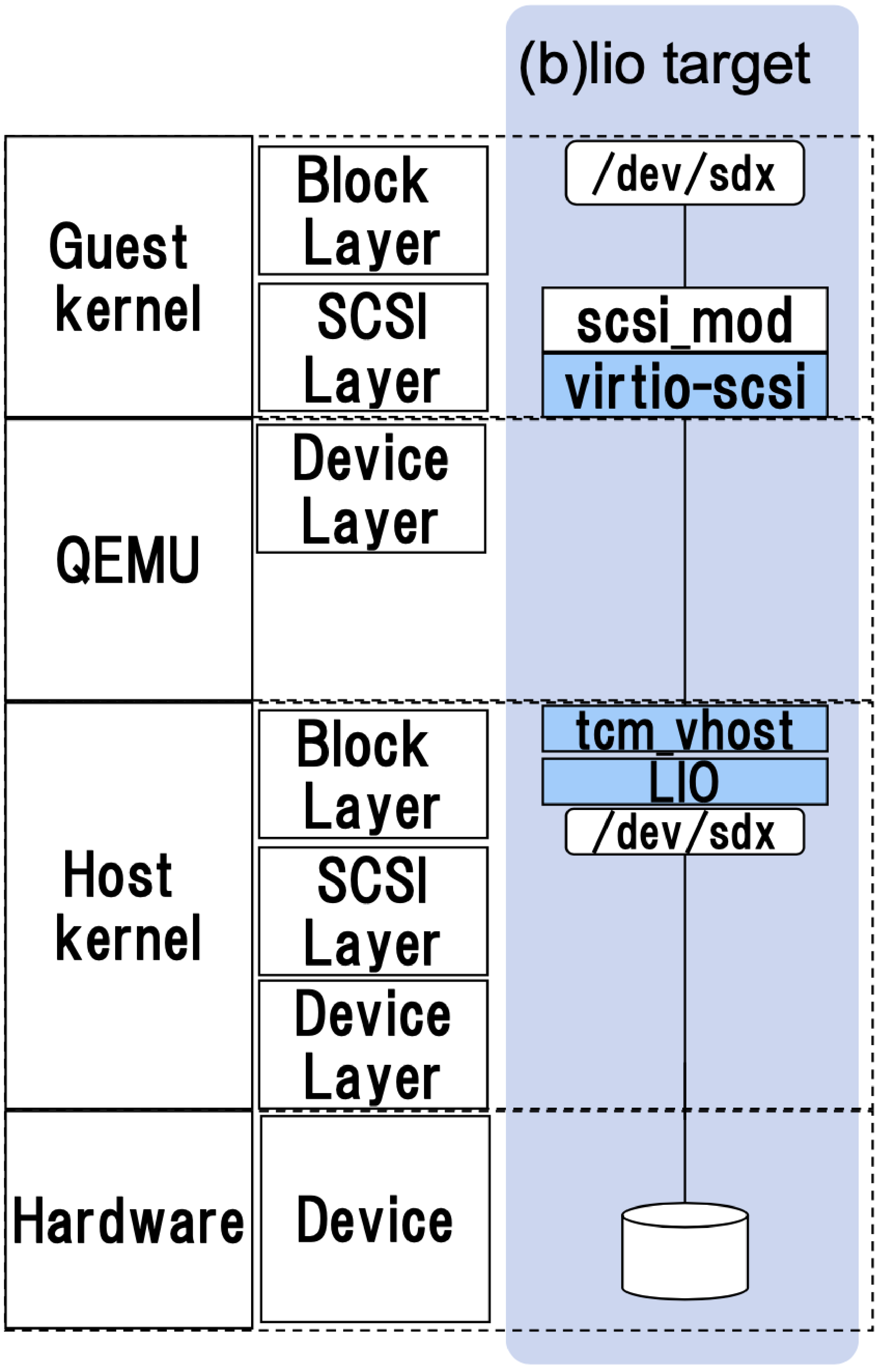
The LIO Linux SCSI Target implements a generic SCSI target that provides remote access to most data storage device types over all prevalent storage fabrics and protocols. LIO neither directly accesses data nor does it directly communicate with applications. LIO provides a fabric-independent and fabric-transparent abstraction for the semantics of numerous data storage device types.

The LIO SCSI target engine is independent of specific fabric modules or backstore types. Thus, LIO supports mixing and matching any number of fabrics and backstores at the same time. The LIO SCSI target engine implements a comprehensive SPC-3/SPC-4 feature set with support for high-end features, including SCSI-3/SCSI-4 Persistent Reservations (PRs), SCSI-4 Asymmetric Logical Unit Assignment (ALUA), VMware vSphere APIs for Array Integration (VAAI), T10 DIF, etc.

The concept of a SCSI target isn't narrowly restricted to physical devices on a SCSI bus, but instead provides a generalized model for all receivers on a logical SCSI fabric. This includes SCSI sessions across interconnects with no physical SCSI bus at all. Conceptually, the SCSI target provides a generic block storage service or server in this scenario.


Target Architecture
CLI
GUI
Library and API (Local and Remote)
Unified Target
Generic Target Engine
SPC-3/4 SCSI Core
Clustering support (PRs, ALUA, Referrals, Fencing)
Smart Array Offloads (VAAI, ODX)
Storage Management Engine
Manage physical and virtual storage resources
Memory allocation and memory map
RDMA buffer management or internal allocation
Fabric Modules
iSCSI
FC/FCoE
InfiniBand SRP/iSER
NTB
vHost
Storage Modules
File
Block
Raw

Now it is easier to understand QEMU backend for LIO:

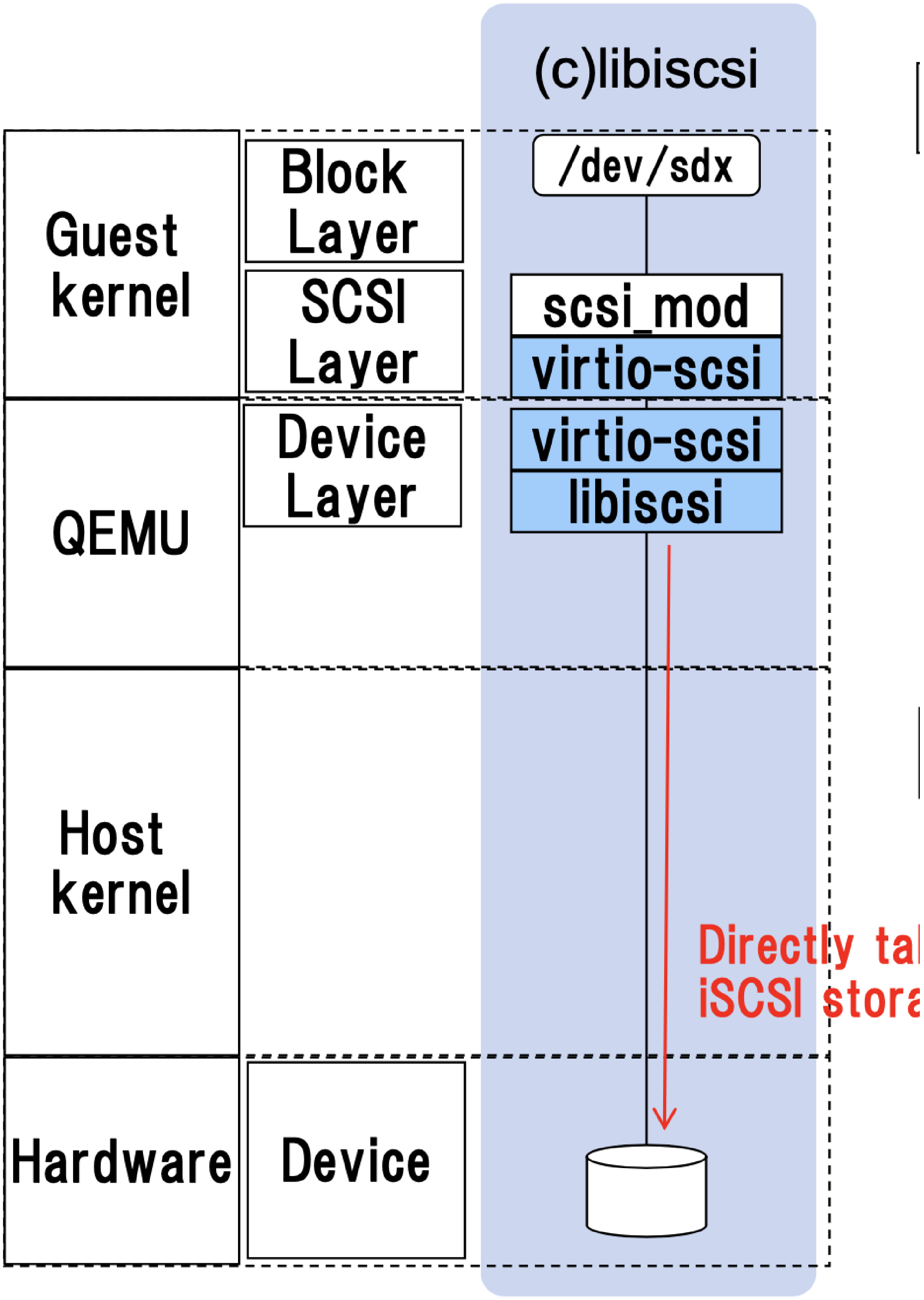

(b)lio target
Guest kernel
Block Layer
SCSI Layer
/dev/sdx
scsi_mod
virtio-scsi
QEMU
Device Layer
Host kernel
Block Layer
SCSI Layer
Device Layer
tcm_vhost
LIO
/dev/sdx
Hardware
Device

As you can see, this virtio-scsi method is close to what virtio-net/vhost-net schema is for a networking device. The vdisk communicates directly with in-kernel LIO subsystem after initially setup by QEMU.

> Analogy: virtio-net <-> vhost-net

iii. virtio-scsi: libiscsi target SCSI handling

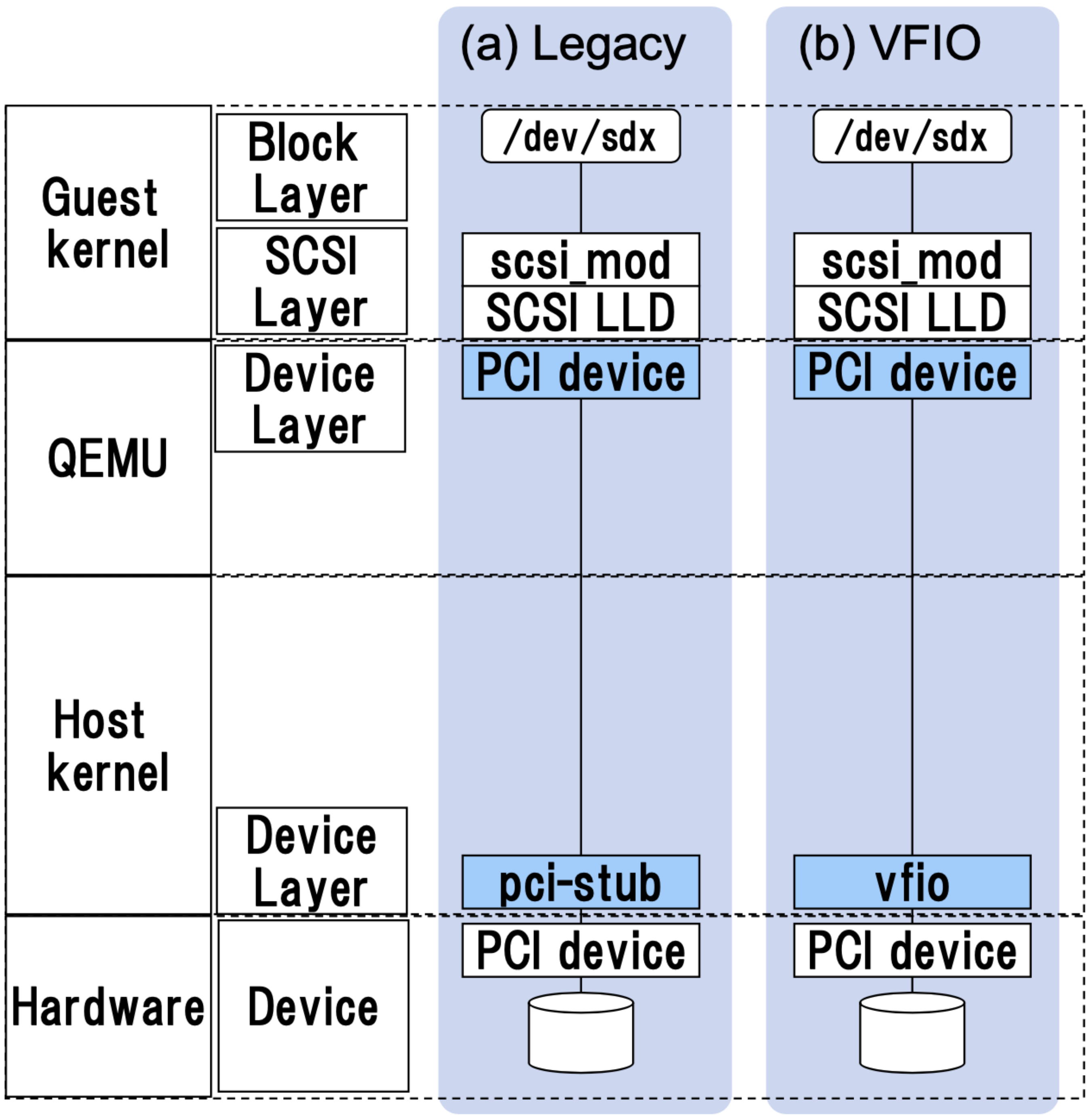
Differently from the LIO target approach, which allows the virtual machine to communicate directly to the HostOS kernel LIO subsystem, by using libiscsi support within QEMU you are making the QEMU emulation process to do the SCSI I/O on your vdisk's behalf:



> Network analogy: virtio-net

1.e.3-) SCSI dev assignment

A vdisk backed by a real SCSI device can also be achieved using VFIO, just as explained in **Device Assignment Methods**.



2-) QEMU Internals

2.a-) Networking

2.a.1-) User Networking - SLIP

This is the default networking backend and generally is the easiest to use. It does not require root / Administrator privileges. It has the following limitations:

There is a lot of overhead so the performance is poor in general, ICMP traffic does not work (so you cannot use ping within a guest) on Linux hosts, ping does work from within the guest, but it needs initial setup by root (once per host) -- see the steps below the guest is not directly accessible from the host or the external network User Networking is implemented using "SLIRP", which provides a full TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network.

> This is usually not used in production environments, nor in the cases being shown here, BUT SLIRP networking CVE will be explained further in this document.

2.a.2-) Networking - TUN/TAP scenario

In one side we have the real NIC, on the other side the tun/tap devices: virtual point-to-point network devices that userspace apps can use to exchange packets:

- tap = layer 2 (ethernet frames) tun = layer 3 (IP packets)

When the tun kernel module is loaded it creates a special device /dev/net/tun. A process can create a tap device opening it and sending special ioctl commands to it. The new tap device has a name in the /dev filesystem and another process can open it, send and receive Ethernet frames.

2.b-) Common Inter-Process Communication (IPC) methods

Unix sockets are one of the ways to do IPC: server binds a socket to a path in the filesystem so a client can connect to it and exchange messages. Unix sockets are also capable of exchanging file descriptors among 2 different processes.

An eventfd is a lighter IPC: while sockets allow to send and receive data, eventfd is basically an integer that might be changed by a producer to signalize a consumer to poll and read data somewhere else. It works as a wait/notify mechanism.

Shared memory is, like the name says, a portion of the OS memory reserved so two or more processes can exchange data from the same pages: one process writes and affects subsequent reads from another process accessing the same memory pages.

2.c-) Virtio specification

Virtqueues are the mechanism for bulk data transport on VIRTIO devices. Each device can have zero or more virtqueues. It is a queue of guest-allocated buffers that the host interacts with (rw). In addition, the VIRTIO specification also defines bi-directional notifications:

- Available Buffer Notification: Used by the driver (guest) to signal there are buffers ready to be processed by the device (QEMU process)

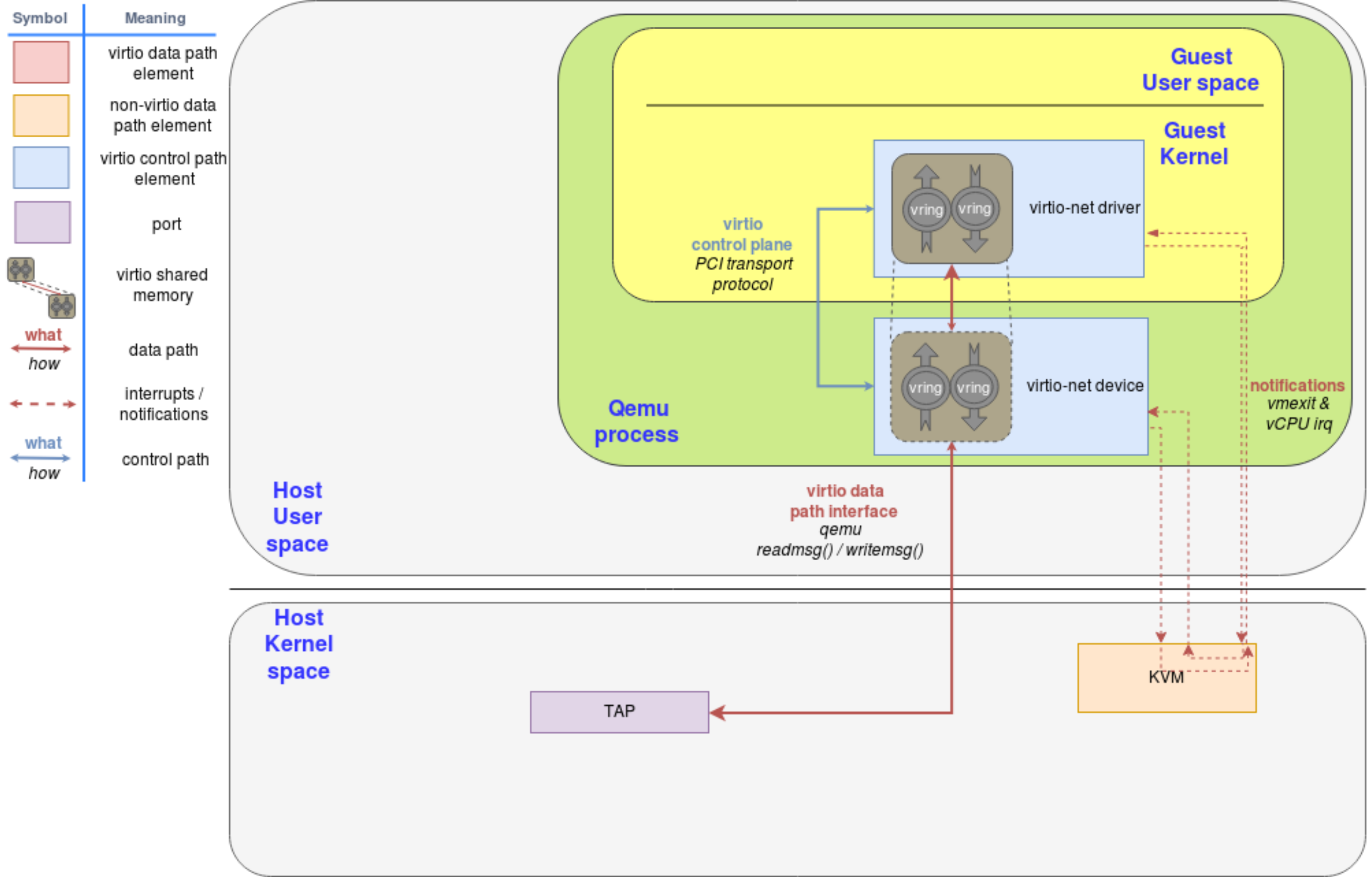
- Used Buffer Notification: Used by the device (QEMU process) to signal driver (guest) it has finished processing some buffers.

In the PCI case, the guest sends the available buffer notification by writing to a specific memory address, and the device (QEMU) uses a vCPU interrupt to send the used buffer notification.

The VIRTIO specification also allows the notifications to be enabled or disabled dynamically. That way, devices and drivers can batch buffer notifications or even actively poll for new buffers in virtqueues (busy polling). This approach is better suited for high traffic rates.

In summary, the VIRTIO driver interface exposes:

- Device’s feature bits (which device and guest have to negotiate)
- Status bits
- Configuration space (that contains device specific information, like MAC address)
- Notification system (configuration changed, buffer available, buffer used)
- Zero or more virtqueues
- Transport specific interface to the device



The VIRTIO network device is a virtual ethernet card, and it supports multiqueue for TX/RX. Empty buffers are placed in N virtqueues for receiving packets, and outgoing packets are enqueued into another N virtqueues for transmission.

Another virtqueue is used for driver-device communication outside of the data plane, like to control advanced filtering features, settings like the mac address, or the number of active queues. As a physical NIC, the VIRTIO device supports features such as many offloadings, and can let the real host's device do them.

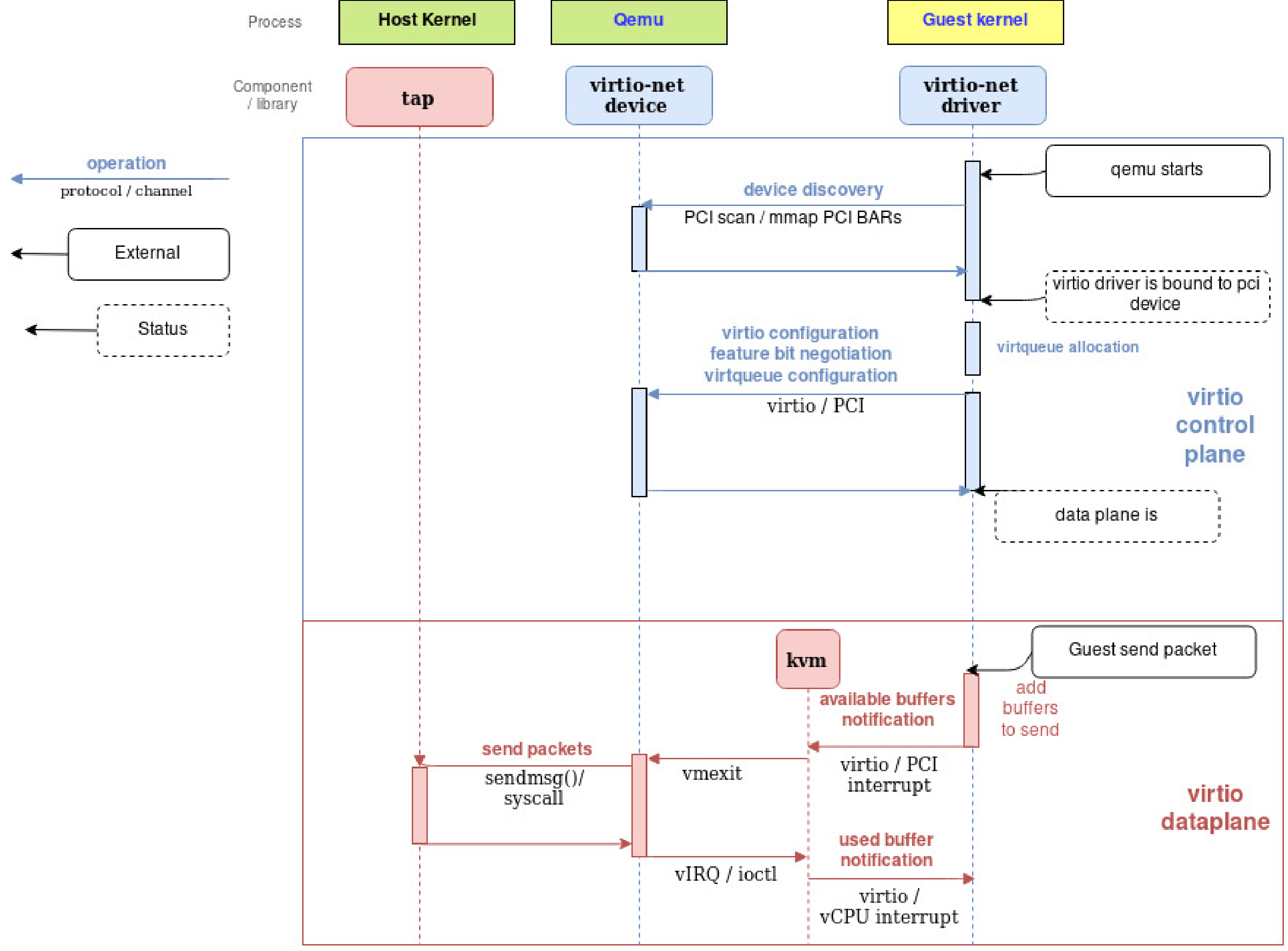
To send a packet, the driver sends to the device a buffer that includes metadata information such as desired offloadings for the packet, followed by the packet frame to transmit. The driver can also split the buffer into multiple gather entries, e.g. it can split the metadata header from the packet frame.

These buffers are managed by the driver and mapped by the device. In this case the device is "inside" the hypervisor. Since the hypervisor (QEMU) has access to all the guests' memory it is capable of locating the buffers and reading or writing them.

The following flow diagram shows the virtio-net device configuration and the sending of a packet using virtio-net driver, that communicates with the virtio-net device over PCI. After filling the packet to be sent, it triggers an "available buffer notification", returning the control to QEMU so it can send the packet through the TAP device.

QEMU then notifies the guest that the buffer operation (reading or writing) is done, and it does that by placing the data in the virtqueue and sending a used notification event, triggering an interruption in the guest vCPU.

The process of receiving a packet is similar to that of sending it. The only difference is that, in this case, empty buffers are pre-allocated by the guest and made available to the device so it can write the incoming data to them.



2.d-) Vhost protocol

The vhost-net is a kernel driver that implements the handler side of the vhost protocol. In this implementation, QEMU and the vhost-net kernel driver (handler) use ioctls to exchange vhost messages and a couple of eventfd-like file descriptors called irqfd and ioeventfd are used to exchange notifications with the guest.

When vhost-net kernel driver is loaded, it exposes a character device on /dev/vhost-net. When QEMU is launched with vhost-net support it opens it and initializes the vhost-net instance with several ioctl(2) calls. These are necessary to associate the hypervisor process with the vhost-net instance, prepare for VIRTIO feature negotiation and pass the guest physical memory mapping to the vhost-net driver.

During the initialization the vhost-net kernel driver creates a kernel thread called vhost-$pid, where $pid is the hypervisor process pid. This thread is called the "vhost worker thread". A tap device is still used to communicate the VM with the host, but now the worker thread handles the I/O events: it polls for driver notifications or tap events, and forwards data.

QEMU allocates one eventfd and registers it to both vhost and KVM in order to achieve the notification bypass. The vhost-$pid kernel thread polls it, and KVM writes to it when the guest writes in a specific address. This mechanism is named ioeventfd (warns host). This way, a simple read/write operation to a specific guest memory address does not need to go through the expensive QEMU process wakeup (a context-switch) and can be routed to the vhost worker kernel thread directly. This also has the advantage of being asynchronous, no need for the vCPU to stop (so no need to do an immediate context switch).

On the other hand, QEMU allocates another eventfd (warns guest) and registers it to both KVM and vhost for direct vCPU interruption injection. This mechanism is called irqfd, and it allows any process in the host to inject vCPU interrupts to the guest by writing to it, with the same advantages (asynchronous, no need for immediate context switching, etc).

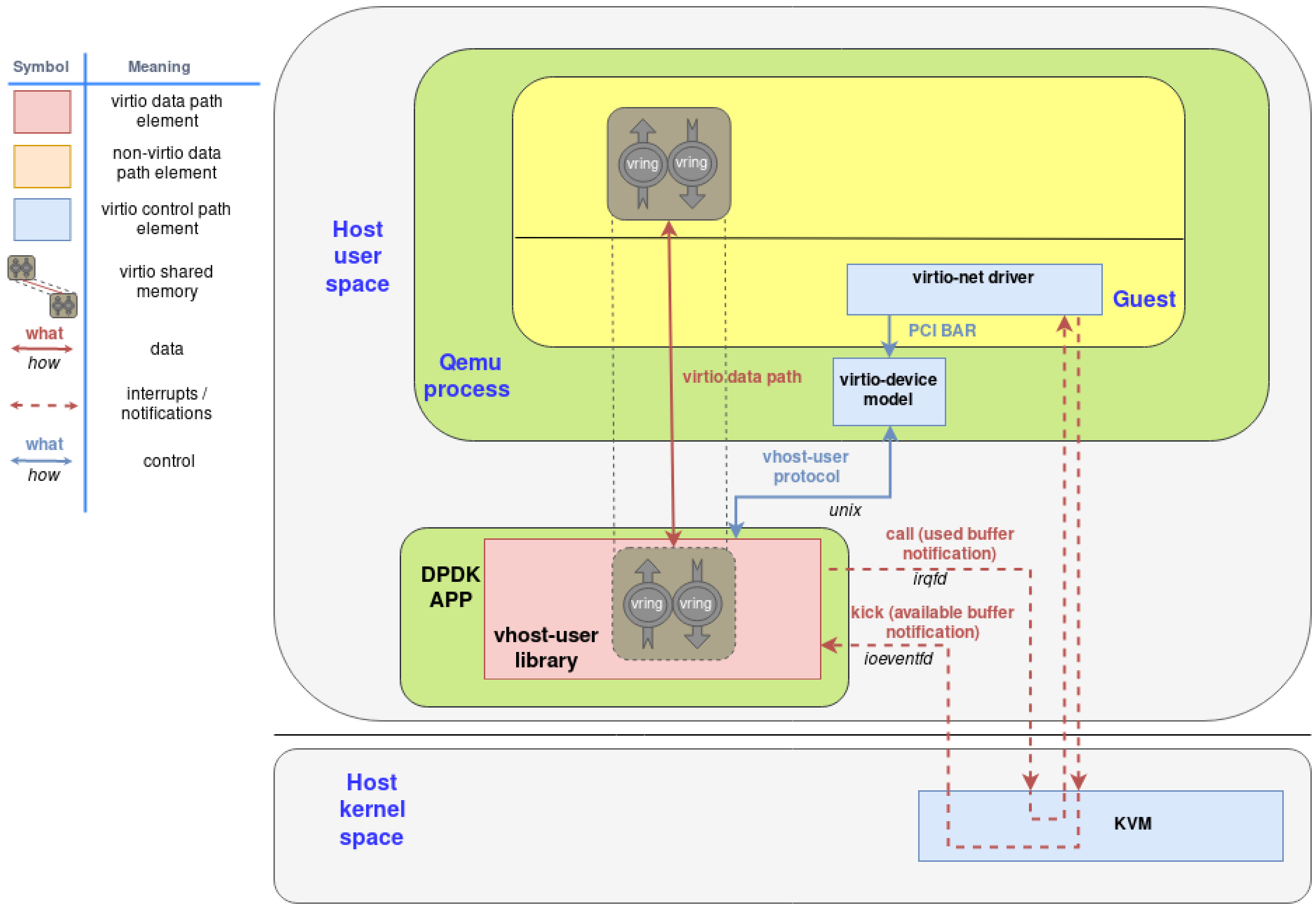
Note: changes in the VIRTIO packet processing backend are completely transparent to the guest who still uses the standard VIRTIO interface.

2.d.1-) The vhost-user protocol

As we've seen, the vhost protocol is a set of messages and mechanisms designed to offload the VIRTIO datapath processing from QEMU (the primary, that wants to offload packet processing) to an external element (the handler, that configures the VIRTIO rings and does the actual packet processing.).

After talking about the vhost-net approach, we can now explore the vhost-user library. This library, built in DPDK, is a userspace implementation of the vhost protocol that allows QEMU to offload the VIRTIO device packet processing to any DPDK application (such as Open vSwitch).

The main difference between the vhost-user library and the vhost-net kernel driver is the communication channel. While the vhost-net kernel driver implements this channel using ioctls, the vhost-user library defines the structure of messages that are sent over a unix socket.



A few points to mention on this diagram:

The VIRTIO memory region is initially allocated by the guest. Corresponding VIRTIO driver interacts with the VIRTIO device normally (through the PCI BARs interface defined in the VIRTIO specification.). The virtio-device-model (inside QEMU) uses the vhost-user protocol to configure the vhost-user library, as well as setting the irqfd and ioeventfd file descriptors. The VIRTIO memory region that was allocated by the guest is mapped (using mmap syscall) by the vhost-user library, i.e: the DPDK application.

> The result is that the DPDK application can read and write packets directly to and from guest memory and use the irqfd and the ioeventfd mechanisms to notify with the guest directly.

> Like explained at "VFIO assignments: vIOMMU support" session, this approach requires vIOMMU to be enabled in the virtual machine and this will be explored further in "QEMU securiRemy Lacroixty" session.

3-) QEMU security surface

To better understand the nature of possible vulnerabilities, escapes and/or security matters in QEMU/KVM execution, we first need to put all the information given so far together.

3.a-) QEMU and OS intersection

This item will be talking about all possible entry/exit points during a QEMU execution. The idea here is not to explicitly talk about vulnerabilities in each of them, but first to create a list of all entry/exit points so we can dedicate to security related observations in next sessions.

As we've seen previously, QEMU running with KVM acceleration will continue to emulate some devices and/or chipsets. For all subsequent comments this document will be considering the following Virtual Machine being executed:

```
<domstatus state='running' reason='unpaused' pid='60547'>
  <monitor path='/var/lib/libvirt/QEMU/domain-18-k8s_749f10a2540344d5/monitor.sock' type='unix'/>
  <vcpus>
    <vcpu id='0' pid='60568'/>
    <vcpu id='1' pid='60569'/>
    <vcpu id='2' pid='60570'/>
    <vcpu id='3' pid='60571'/>
  </vcpus>
  <QEMUCaps>
    <flag name='KVM'/>
    <flag name='no-hpet'/>
    <flag name='virtio-tx-alg'/>
    <flag name='virtio-blk-pci.ioeventfd'/>
    <flag name='virtio-blk-pci.event_idx'/>
    <flag name='virtio-net-pci.event_idx'/>
    <flag name='piix3-usb-uhci'/>
    <flag name='piix4-usb-uhci'/>
    <flag name='vt82c686b-usb-uhci'/>
    <flag name='usb-hub'/>
    <flag name='no-acpi'/>
    <flag name='virtio-blk-pci.scsi'/>
    <flag name='scsi-disk.channel'/>
    <flag name='scsi-block'/>
    <flag name='dump-guest-memory'/>
    <flag name='virtio-scsi-pci'/>
    <flag name='blockio'/>
    <flag name='disable-s3'/>
    <flag name='disable-s4'/>
    <flag name='ide-drive.wwn'/>
    <flag name='scsi-disk.wwn'/>
    <flag name='seccomp-sandbox'/>
    <flag name='reboot-timeout'/>
    <flag name='vnc'/>
    <flag name='cirrus-vga'/>
    <flag name='device-video-primary'/>
    <flag name='nbd-server'/>
    <flag name='virtio-rng'/>
    <flag name='rng-random'/>
    <flag name='rng-egd'/>
    <flag name='pci-bridge'/>
    <flag name='mem-merge'/>
    <flag name='drive-discard'/>
    <flag name='i440fx-pci-hole64-size'/>
    <flag name='KVM-pit-lost-tick-policy'/>
    <flag name='boot-strict'/>
    <flag name='usb-kbd'/>
    <flag name='msg-timestamp'/>
    <flag name='active-commit'/>
    <flag name='change-backing-file'/>
    <flag name='memory-backend-ram'/>
    <flag name='numa'/>
    <flag name='memory-backend-file'/>
    <flag name='rtc-reset-reinjection'/>
    <flag name='splash-timeout'/>
    <flag name='iothread'/>
    <flag name='migrate-rdma'/>
    <flag name='drive-iotune-max'/>
    <flag name='pc-dimm'/>
    <flag name='machine-vmport-opt'/>
    <flag name='aes-key-wrap'/>
    <flag name='dea-key-wrap'/>
    <flag name='vhost-user-multiqueue'/>
    <flag name='migration-event'/>
    <flag name='virtio-net'/>
    <flag name='gic-version'/>
    <flag name='incoming-defer'/>
    <flag name='chardev-file-append'/>
    <flag name='vserport-change-event'/>
    <flag name='virtio-balloon-pci.deflate-on-oom'/>
    <flag name='chardev-logfile'/>
    <flag name='debug-threads'/>
    <flag name='secret'/>
    <flag name='virtio-scsi-pci.iothread'/>
    <flag name='name-guest'/>
    <flag name='drive-detect-zeroes'/>
    <flag name='tls-creds-x509'/>
    <flag name='smm'/>
    <flag name='virtio-pci-disable-legacy'/>
    <flag name='query-hotpluggable-cpus'/>
    <flag name='virtio-net.rx_queue_size'/>
    <flag name='drive-iotune-max-length'/>
    <flag name='query-qmp-schema'/>
    <flag name='gluster.debug_level'/>
    <flag name='vhost-scsi'/>
    <flag name='drive-iotune-group'/>
    <flag name='query-cpu-model-expansion'/>
    <flag name='virtio-net.host_mtu'/>
    <flag name='query-cpu-definitions'/>
    <flag name='block-write-threshold'/>
    <flag name='query-named-block-nodes'/>
    <flag name='cpu-cache'/>
    <flag name='kernel-irqchip'/>
    <flag name='kernel-irqchip.split'/>
    <flag name='virtio.iommu_platform'/>
    <flag name='virtio.ats'/>
    <flag name='loadparm'/>
    <flag name='vnc-multi-servers'/>
    <flag name='virtio-net.tx_queue_size'/>
    <flag name='chardev-reconnect'/>
    <flag name='vxhs'/>
    <flag name='virtio-blk.num-queues'/>
    <flag name='vmcoreinfo'/>
    <flag name='numa.dist'/>
    <flag name='disk-share-rw'/>
    <flag name='iscsi.password-secret'/>
    <flag name='isa-serial'/>
    <flag name='dump-completed'/>
    <flag name='qcow2-luks'/>
    <flag name='seccomp-blacklist'/>
    <flag name='query-cpus-fast'/>
    <flag name='disk-write-cache'/>
    <flag name='nbd-tls'/>
    <flag name='pr-manager-helper'/>
    <flag name='qom-list-properties'/>
    <flag name='memory-backend-file.discard-data'/>
    <flag name='sdl-gl'/>
    <flag name='screendump_device'/>
    <flag name='blockdev-del'/>
    <flag name='vmgenid'/>
    <flag name='vhost-vsock'/>
    <flag name='chardev-fd-pass'/>
    <flag name='egl-headless'/>
    <flag name='blockdev'/>
    <flag name='memory-backend-memfd'/>
    <flag name='memory-backend-memfd.hugetlb'/>
    <flag name='iothread.poll-max-ns'/>
    <flag name='egl-headless.rendernode'/>
    <flag name='memory-backend-file.align'/>
    <flag name='memory-backend-file.pmem'/>
    <flag name='scsi-disk.device_id'/>
    <flag name='virtio-pci-non-transitional'/>
    <flag name='overcommit'/>
    <flag name='query-current-machine'/>
    <flag name='bitmap-merge'/>
    <flag name='nbd-bitmap'/>
    <flag name='x86-max-cpu'/>
    <flag name='cpu-unavailable-features'/>
    <flag name='canonical-cpu-features'/>
    <flag name='migration-file-drop-cache'/>
    <flag name='ramfb'/>
    <flag name='blockdev-file-dynamic-auto-read-only'/>
    <flag name='savevm-monitor-nodes'/>
    <flag name='drive-nvme'/>
  </QEMUCaps>
  <devices>
    <device alias='rng0'/>
    <device alias='virtio-disk2'/>
    <device alias='virtio-disk1'/>
    <device alias='virtio-disk0'/>
    <device alias='video0'/>
    <device alias='serial0'/>
    <device alias='net0'/>
    <device alias='balloon0'/>
    <device alias='usb'/>
  </devices>
  <libDir path='/var/lib/libvirt/QEMU/domain-18-k8s_749f10a2540344d5'/>
  <channelTargetDir path='/var/lib/libvirt/QEMU/channel/target/domain-18-k8s_749f10a2540344d5'/>
  <chardevStdioLogd/>
  <rememberOwner/>
  <allowReboot value='yes'/>
  <nodename index='4'/>
  <blockjobs active='no'/>
  <agentTimeout>-2</agentTimeout>
  <domain type='KVM' id='18'>
    <name>k8s_749f10a2540344d58acde50791adb38e_7212_34a072e8-219b-4482-b70f-ab4ade4c58f3</name>
    <uuid>6814edd9-cd66-4256-a603-6f6591841c8c</uuid>
    <memory unit='KiB'>16781312</memory>
    <currentMemory unit='KiB'>16781312</currentMemory>
    <vcpu placement='static' cpuset='17,19,49,51'>4</vcpu>
    <cputune>
      <vcpupin vcpu='0' cpuset='17,49'/>
      <vcpupin vcpu='1' cpuset='17,49'/>
      <vcpupin vcpu='2' cpuset='19,51'/>
    <vcpupin vcpu='3' cpuset='19,51'/>
    <emulatorpin cpuset='17,19,49,51'/>
  </cputune>
  <numatune>
    <memory mode='strict' nodeset='1'/>
    <memnode cellid='0' mode='strict' nodeset='1'/>
  </numatune>
  <resource>
    <partition>/machine</partition>
  </resource>
  <sysinfo type='smbios'>
    <system>
      <entry name='family'>7212_34a072e8-219b-4482-b70f-ab4ade4c58f3</entry>
    </system>
    <baseBoard>
      <entry name='manufacturer'>KVM:2dcee5e5e43726c4fc99fb8db99e2b7b76bca613</entry>
      <entry name='asset'>34a072e8-219b-4482-b70f-ab4ade4c58f3</entry>
    </baseBoard>
    <chassis>
      <entry name='manufacturer'></entry>
      <entry name='asset'></entry>
    </chassis>
  </sysinfo>
  <os>
    <type arch='x86_64' machine='pc-i440fx-4.2'>hvm</type>
    <boot dev='hd'/>
    <smbios mode='sysinfo'/>
  </os>
  <features>
    <acpi/>
  </features>
  <cpu mode='custom' match='exact' check='full'>
    <model fallback='forbid'>Broadwell-IBRS</model>
    <vendor>Intel</vendor>
    <topology sockets='1' cores='2' threads='2'/>
    <feature policy='require' name='vme'/>
    <feature policy='require' name='ss'/>
    <feature policy='require' name='vmx'/>
    <feature policy='require' name='f16c'/>
    <feature policy='require' name='rdrand'/>
    <feature policy='require' name='hypervisor'/>
    <feature policy='require' name='arat'/>
    <feature policy='require' name='tsc_adjust'/>
    <feature policy='require' name='stibp'/>
    <feature policy='require' name='ssbd'/>
    <feature policy='require' name='xsaveopt'/>
    <feature policy='require' name='pdpe1gb'/>
    <feature policy='require' name='abm'/>
    <numa>
      <cell id='0' cpus='0-3' memory='16781312' unit='KiB'/>
    </numa>
  </cpu>
  <clock offset='utc'>
    <timer name='rtc' tickpolicy='catchup'/>
  </clock>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <disk type='file' device='disk'>
      <driver name='QEMU' type='qcow2' cache='writethrough' error_policy='report'/>
      <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.qcow2' index='3'>
        <privateData>
          <nodenames>
            <nodename type='storage' name='libvirt-3-storage'/>
            <nodename type='format' name='libvirt-3-format'/>
          </nodenames>
        </privateData>
      </source>
      <backingStore type='file' index='4'>
        <format type='qcow2'/>
        <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.qcow2'>
          <privateData>
            <nodenames>
              <nodename type='storage' name='libvirt-4-storage'/>
              <nodename type='format' name='libvirt-4-format'/>
            </nodenames>
          </privateData>
        </source>
        <backingStore/>
      </backingStore>
      <target dev='vda' bus='virtio'/>
      <iotune>
        <total_bytes_sec>49152000</total_bytes_sec>
        <total_iops_sec>3000</total_iops_sec>
        <total_bytes_sec_max>49152000</total_bytes_sec_max>
        <total_iops_sec_max>3000</total_iops_sec_max>
      </iotune>
      <serial>7212-66536caf-30c4-4f8c-984f-c41749ae3f9e-pzgpn</serial>
      <alias name='virtio-disk0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
      <privateData>
        <qom name='/machine/peripheral/virtio-disk0/virtio-backend'/>
      </privateData>
    </disk>
    <disk type='file' device='disk'>
      <driver name='QEMU' type='raw' cache='writethrough' error_policy='report'/>
      <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/init-disk' index='2'>
        <privateData>
          <nodenames>
            <nodename type='storage' name='libvirt-2-storage'/>
            <nodename type='format' name='libvirt-2-format'/>
          </nodenames>
        </privateData>
      </source>
      <backingStore/>
      <target dev='vdb' bus='virtio'/>
      <iotune>
        <total_bytes_sec>16384000</total_bytes_sec>
        <total_iops_sec>1000</total_iops_sec>
        <total_bytes_sec_max>16384000</total_bytes_sec_max>
        <total_iops_sec_max>1000</total_iops_sec_max>
      </iotune>
      <serial>cloud-init-7212_34a072e8-219b-4482-b70f-ab4ade4c58f3-bphhv</serial>
      <alias name='virtio-disk1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
      <privateData>
        <qom name='/machine/peripheral/virtio-disk1/virtio-backend'/>
      </privateData>
    </disk>
    <disk type='file' device='disk'>
      <driver name='QEMU' type='raw' cache='writethrough' error_policy='report'/>
      <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/swap' index='1'>
        <privateData>
          <nodenames>
            <nodename type='storage' name='libvirt-1-storage'/>
            <nodename type='format' name='libvirt-1-format'/>
          </nodenames>
        </privateData>
      </source>
      <backingStore/>
      <target dev='vdc' bus='virtio'/>
      <iotune>
        <total_bytes_sec>16384000</total_bytes_sec>
        <total_iops_sec>1000</total_iops_sec>
        <total_bytes_sec_max>16384000</total_bytes_sec_max>
        <total_iops_sec_max>1000</total_iops_sec_max>
      </iotune>
      <serial>cloud-init-</serial>
      <alias name='virtio-disk2'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
      <privateData>
        <qom name='/machine/peripheral/virtio-disk2/virtio-backend'/>
      </privateData>
    </disk>
    <controller type='usb' index='0' model='piix3-uhci'>
      <alias name='usb'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
    </controller>
    <controller type='pci' index='0' model='pci-root'>
      <alias name='pci.0'/>
    </controller>
    <interface type='direct'>
      <mac address='aa:bb:cc:dd:ee:ff'/>
      <source dev='if0aabbccaabbc' mode='passthrough'/>
      <bandwidth>
        <inbound average='1000000' peak='1000000'/>
      </bandwidth>
      <target dev='macvtap0'/>
      <model type='virtio'/>
      <driver queues='3'/>
      <alias name='net0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
    <serial type='pty'>
      <source path='/dev/pts/0'/>
      <target type='isa-serial' port='0'>
        <model name='isa-serial'/>
      </target>
      <alias name='serial0'/>
    </serial>
    <console type='pty' tty='/dev/pts/0'>
      <source path='/dev/pts/0'/>
      <target type='serial' port='0'/>
      <alias name='serial0'/>
    </console>
    <input type='mouse' bus='ps2'>
      <alias name='input0'/>
    </input>
```

```
   <input type='keyboard' bus='ps2'>
    <alias name='input1'/>
   </input>
   <graphics type='vnc' port='5900' autoport='yes' websocketGenerated='no' listen='0.0.0.0' passwd='******'>
    <listen type='address' address='0.0.0.0' fromConfig='0' autoGenerated='no'/>
   </graphics>
   <video>
    <model type='cirrus' vram='16384' heads='1' primary='yes'/>
    <alias name='video0'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
   </video>
   <memballoon model='virtio'>
    <stats period='30'/>
    <alias name='balloon0'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x07' function='0x0'/>
   </memballoon>
   <rng model='virtio'>
    <rate bytes='1234' period='2000'/>
    <backend model='random'>/dev/urandom</backend>
    <alias name='rng0'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x08' function='0x0'/>
   </rng>
  </devices>
  <seclabel type='dynamic' model='apparmor' relabel='yes'>
   <label>libvirt-6814edd9-cd66-4256-a603-6f6591841c8c</label>
   <imagelabel>libvirt-6814edd9-cd66-4256-a603-6f6591841c8c</imagelabel>
  </seclabel>
  <seclabel type='dynamic' model='dac' relabel='yes'>
   <label>+64055:+116</label>
   <imagelabel>+64055:+116</imagelabel>
  </seclabel>
 </domain>
</domstatus>
```

3.b-) QEMU vCPU assignment & side-channel observations

This document won't concentrate many efforts in explaining side channel attacks and its mitigations, as those are HW related and would require, perhaps, many documents to be explained. Nevertheless, some explanation might be worth, specially regarding why avoiding vCPUs being scheduled in the same real HW cores/threads could be a good idea.

You will find more information about this topic HERE and in all documents listed there as references.

For now, all you need to know is: a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself. Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.

General classes of side channel attack do exist, including power-monitoring, electromagnetic and acoustic analysis, etc. For our case, virtualization world, the important classes are the ones related to CPU internals observations.

The following CVEs (and more) have been discovered on this topic:

Side Channel Attacks - Spectre and Meltdown

- CVE-2017-5753 - Bounds Check Bypass (Variant 1 / Spectre)
- CVE-2017-5715 - Branch Target Injection (Variant 2 / Spectre)
- CVE-2017-5754 - Rogue Data Cache Load (Variant 3 / Meltdown)

Side Channel Attacks - Others

- CVE-2018-3665 - Lazy FP Save/Restore (LazyFP)
- CVE-2018-3693 - Bounds Check Bypass Store (Variant (or Spectre) 1.1 and 1.2 / BCBS)
- CVE-2018-3640 - Rogue System Register Read (RSRE / Variant 3a)
- CVE-2018-3639 - Speculative Store Bypass (SSB / Variant 4 / Spectre-NG)

L1 Terminal Fault (L1TF)

- CVE-2018-3615 - Intel SGX (Software Guard Extensions) (Foreshadow / L1TF)
- CVE-2018-3620 - Operating Systems and System Management Mode (Fault-OS / SMM) (L1TF)
- CVE-2018-3646 - Virtualization Extensions (L1TF)

Microarchitectural Data Sampling (MDS)

- CVE-2018-12126 - Microarchitectural Store Buffer Data Sampling (MSBDS / Fallout)
- CVE-2018-12127 - Microarchitectural Load Port Data Sampling (MLPDS / RIDL)
- CVE-2018-12130 - Microarchitectural Fill Buffer Data Sampling (MFBDS / ZombieLoad)
- CVE-2019-11091 - Microarchitectural Data Sampling Uncacheable Memory (MDSUM)

> This list isn't fully updated to the latest, check official sources for updated information

And this is why it's important to understand risks and mitigations for this class of security vulnerabilities.

If we take as an example the following domain XML definition:

```
<vcpu placement='static' cpuset='17,19,49,51'>4</vcpu>
<cputune>
  <vcpupin vcpu='0' cpuset='17,49'/>
  <vcpupin vcpu='1' cpuset='17,49'/>
  <vcpupin vcpu='2' cpuset='19,51'/>
  <vcpupin vcpu='3' cpuset='19,51'/>
  <emulatorpin cpuset='17,19,49,51'/>
</cputune>
<numatune>
  <memory mode='strict' nodeset='1'/>
  <memnode cellid='0' mode='strict' nodeset='1'/>
</numatune>
...
<cpu mode='custom' match='exact' check='full'>
  <model fallback='forbid'>Broadwell-IBRS</model>
  <vendor>Intel</vendor>
  <topology sockets='1' cores='2' threads='2'/>
  <feature policy='require' name='vme'/>
  <feature policy='require' name='ss'/>
  <feature policy='require' name='vmx'/>
  <feature policy='require' name='f16c'/>
  <feature policy='require' name='rdrand'/>
  <feature policy='require' name='hypervisor'/>
  <feature policy='require' name='arat'/>
  <feature policy='require' name='tsc_adjust'/>
  <feature policy='require' name='stibp'/>
  <feature policy='require' name='ssbd'/>
  <feature policy='require' name='xsaveopt'/>
  <feature policy='require' name='pdpe1gb'/>
  <feature policy='require' name='abm'/>
  <numa>
    <cell id='0' cpus='0-3' memory='16781312' unit='KiB'/>
  </numa>
</cpu>
```

We can see that libvirt & QEMU will operate together in order to create 4 x OS processes acting as vCPUs for the virtual machine being created. This can also be seen at the top of the descriptive XML file (after machine is already running):

```
<vcpus>
  <vcpu id='0' pid='60568'/>
  <vcpu id='1' pid='60569'/>
  <vcpu id='2' pid='60570'/>
  <vcpu id='3' pid='60571'/>
</vcpus>
```

Without entering the performance world, trying to stay only in the security effectiveness of this setup, we have to understand why each of those PIDs were placed in specific CPUs. If we take a look at the 'lscpu' output we will understand how NUMA architecture was designed for the host chipset:

```
NUMA node0 CPU(s):   0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54,56,58,60,6
NUMA node1 CPU(s):   1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55,57,59,61,6
```

We still don't know how memory displacement is set within the NUMA domains, but this is already enough for to understand that this machine has 2 x NUMA domains, each one with 1 x socket with 16 x cores (and 2 x threads) each. That will give enough information about CPU sharing among different virtual machines and that information is imperative for us to understand side-channel attack techniques (and how they could affect the environment).

By creating processor sets:

```
<cputune>
  <vcpupin vcpu='0' cpuset='17,49'/>
  <vcpupin vcpu='1' cpuset='17,49'/>
  <vcpupin vcpu='2' cpuset='19,51'/>
  <vcpupin vcpu='3' cpuset='19,51'/>
  <emulatorpin cpuset='17,19,49,51'/>
</cputune>
```

and placing QEMU vCPU and emulation processes (at least 5 x OS processes in this example) in the same processor set (placed in real CPUs #17, #49, #19, #51), libvirt uses the host kernel cgroup facility to provide isolation for these vCPUs: they will only migrate in between the real CPUs defined int the XML.

- vCPUs 0 & 1 will migrate between real CPUs 17 and 49
- vCPUs 2 & 3 will migrate between real CPUs 19 and 51
- QEMU emulation thread will migrate in between the 4 real CPUs

Having only 2 real CPUs available to each vCPU, and those 2 real CPUs shared among the 2 vCPUs as well, tell us something: there is clearly an intent to keep the vCPU scheduled within the same real CPU core. It is likely working as a "2 threads" per 2 vCPUs virtual machine.

We can confirm that by checking topology in sysfs:

```
/sys/devices/system/cpu/cpu17/topology:
$ cat thread_siblings_list
17,49
/sys/devices/system/cpu/cpu19/topology:
$ cat thread_siblings_list
19,51
```

This means that this virtual machine has 4 vCPUs (2 cores and 2 threads) and both of its cores are placed in the same NUMA domain (anything different from this could cause performance problems because of memory latency being bigger in between the NUMA domains).

Why is it important to understand the CPU pinning and sharing concepts ?

Side channel attacks are related to CPU sharing: an attacker tries to discover information from internal (to CPU) cached data by executing other instructions in the CPU right after a task migration (in-off the CPU) happened. Some vulnerabilities are related to boundaries on branch instructions (Spectre / Variant 1), others are related to CPU indirect branches (Spectre / Variant 2).

Particularly for Spectre Variant 2, there are some mitigations that we can mention here that is worth. CPU firmware might have mitigations for this side channel attack type: IBRS, STIBP and IPBP. Some of those terms can be found in the virtual machine definition XML from our example:

```
<cpu mode='custom' match='exact' check='full'>
  <model fallback='forbid'>Broadwell-IBRS</model>
  <vendor>Intel</vendor>
  <topology sockets='1' cores='2' threads='2'/>
  <feature policy='require' name='stibp'/>
  <feature policy='require' name='ssbd'/>
  ...
</cpu>
```

This XML is telling that any CPU not containing the mitigation STIBP will not satisfy the VM needs to be started: The Single Thread Indirect Branch Predictors (STIBP) prevents indirect branch predictions from being controlled by the sibling HW thread. This might also explain why a 1:1 relationship between the CPU HW threads and the vCPU pinnings exist:

```
<cputune>
  <vcpupin vcpu='0' cpuset='17,49'/>
  <vcpupin vcpu='1' cpuset='17,49'/>
  ...
```

By not sharing CPUs #17 and #49 (HW siblings threads) with any workload (just this VM) you could be mitigating risks for this side channel technique. By having the STIBP firmware mitigation you reduce risks of sharing the CPU HW thread between different workloads.

This was just 1 example, for 1 specific CVE and its mitigation. There are many other side channel attacks reported and mitigated already, for multiple architectures, and the intent of this document is only to instruct reader how to better understand the correlation between the mitigation technique, mitigation nomenclature and how QEMU deals with it.

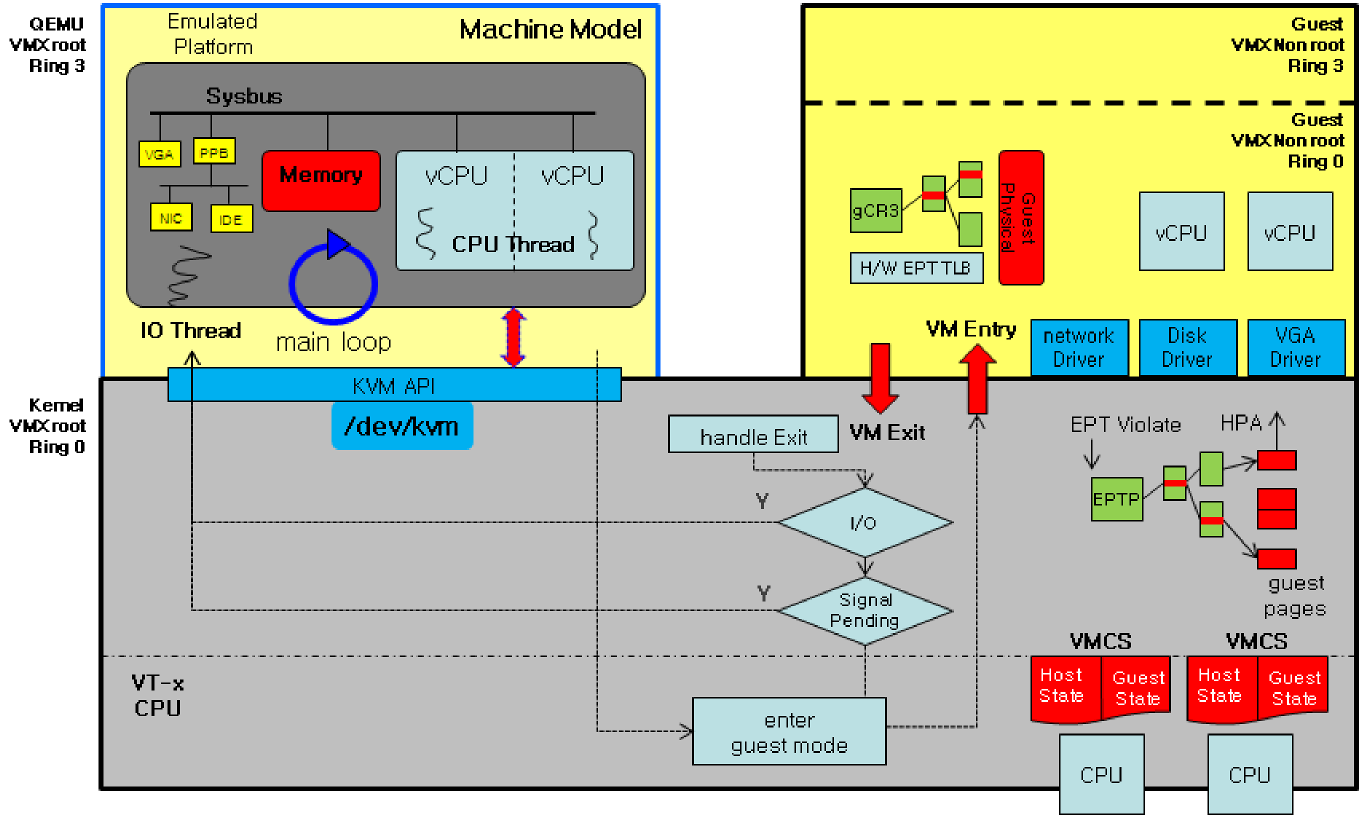
> More details can be found at this place

A quick observation about side channel attacks logging and introspection: it is very hard, if not impossible, to see if a user is targeting a system with those types of techniques. For some cases a CPU instruction introspection would be needed - to realize that a particular CPU part is being speculated - as the workload would not differentiate much from the regular CPU workloads. That is why most of these mitigations are better handled by CPU firmware changes and those also try to cope with performance penalties the mitigations will cause.

> Note: There are also some software only mitigations, like Kernel Page-Table Isolation - KPTI, or just PTI - which is software-only technique that helped in the Meltdown vulnerability.

3.c-) QEMU devices emulation

Whenever QEMU is using KVM as its VMM (virtual machine manager) it does not have to emulate (or intercept) instructions unless some emulation is required. Without entering in too many technical details on how this is done, the important part to understand is this:



Paying special attention to the right side of the picture above, you can see "VM Entry" and "VM Exist" signs. The main idea is this: the KVM module sets the logical vCPU states registers into a specific area and, using an OS process (in name of a vCPU) it calls a "enter virtualization" instruction, that will start running code pointed by the logical vCPU registers (and the vCPU assumes control). The QEMU OS vCPU process will be waiting for this call to return meanwhile the vCPU process is running guest OS instructions.

Now, imagining the guest as a machine, it will be running its kernel and all its user-land code inside those 4 vCPUs - let's say - and causing its OS processes to enter and leave the vCPUs. This won't affect QEMU vCPU processes (4 of them), that will stay waiting the "enter virtualization" instruction to end. If, for any reason, the guest touches memory addresses of an area that was described in the logical vCPU registes as an area that needs emulation... it will cause a VM_EXIT and the QEMU vCPU process will return from the "enter virtualization" call.

QEMU will have to understand the reason for the VM_EXIT to happen - and there is a specific reason for EMULATED I/O need - and deal with the emulation for the memory address change before returning control to the vCPU again, doing another "enter virtualization" call for the same vCPU OS process.

This process can be seen when using tracing tools like systemtap (ebpf, ftrace, etc). Those traces allows you to do something on a software break point (QEMU). With systemtap, a simple script such as:

```
probe process("QEMU-system-x86_64").function("KVM_arch_post_run")
{
  try {
    reason = $cpu->KVM_run->exit_reason;
    if (reason != 2) {
      printf("EXIT reason: %u\n", reason);
    }
    if (reason == 2) {
      port = $cpu->KVM_run->io->port;
      printf("I/O port: %x\n", port);
    }
  } catch (msg) {
    println("error\n", msg)
  }
}
```

will tell you the reason for a vCPU to have exited the 'virtualization function' (Intel VT-x). In the code above, we are focusing in an exist through IO emulation, like being discussed in this topic.

As you might already have thought by now, security issues for emulated devices rise from the fact that the guest OS kernel driver communicates to hypervisor through this emulation. Vulnerabilities here are tightly coupled to QEMU emulation code and could case QEMU to run arbitrary instructions on behalf of a guest, for example.

From the example we are using in this document, the emulation would come from the follow devices being defined to this virtual machine:

```
<devices>
  <controller type='usb' index='0' model='piix3-uhci'>
    <alias name='usb'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
  </controller>
  <controller type='pci' index='0' model='pci-root'>
    <alias name='pci.0'/>
  </controller>
  <serial type='pty'>
    <source path='/dev/pts/0'/>
    <target type='isa-serial' port='0'>
      <model name='isa-serial'/>
    </target>
    <alias name='serial0'/>
  </serial>
  <console type='pty' tty='/dev/pts/0'>
    <source path='/dev/pts/0'/>
    <target type='serial' port='0'/>
    <alias name='serial0'/>
  </console>
  <input type='mouse' bus='ps2'>
    <alias name='input0'/>
  </input>
  <input type='keyboard' bus='ps2'>
    <alias name='input1'/>
  </input>
  <video>
    <model type='cirrus' vram='16384' heads='1' primary='yes'/>
    <alias name='video0'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
  </video>
</devices>
```

From the current list we have the following emulated devices:

- ISA controller
- PS2 controller (mouse and keyboard)
- PCI controller
- USB controller in PCI bus
- ISA Serial Controller
- Cirrus PCI VGA Adapter

> The security surface here relies in how the devices are emulated inside the QEMU code and potential CVEs targeting the device emulation code within QEMU.

If we take the QEMU serial emulation by an example of vulnerability surface:

```
[    0.750547] Serial: 8250/16550 driver, 32 ports, IRQ sharing enabled
```

Anytime the guest writes to its internal ISA address 0x3ff - the old 16-bit internal BUS, being emulated by QEMU as well - using the following instructions:

```
__asm("movw $0x3ff, %dx");   // ttys0 isa i/o address
__asm("movb $0x1f, %al");    // moves ascii unit sep. char to %al
__asm("outb %al, (%dx)");    // writes the char to scratch register (0x3ff)
```

the vCPU (inside guest) responsible for the execution of the I/O instruction will exit the virtualization function (VM_EXIT) on the host so QEMU can emulate the 8250 serial controller HW (the virtualized memory registers) and, right after, return control to the guest vCPU.

If, again, using systemtap, we create a script like:

```
probe process("QEMU-system-x86_64").function("KVM_arch_post_run")
{
  try {
    reason = $cpu->KVM_run->exit_reason;
    if (reason == 2) {
      port = $cpu->KVM_run->io->port;
      dir = $cpu->KVM_run->io->direction;
      size = $cpu->KVM_run->io->size;
      count = $cpu->KVM_run->io->count;

      if (port == 0x3ff) {
          ptr = $cpu->KVM_run;
          offset = $cpu->KVM_run->io->data_offset;
          data = ptr + offset;
          // do something
      }
    }
  } catch {

  }
}
```

> systemtap is capable of probing QEMU-system-x86_64 process internal functions and executing code described above, whenever described probe (function KVM_arch_post_run) is called (similarly to a debugger break point).

We will be able to monitor (trace / debug) each time a single character is written to device "/dev/ttyS0" inside the virtual machine. Each character written to an ISA 8250 controller causes a VM_EXIT of exec'ing vCPU: it gives the real CPU back to the QEMU process and it will emulate the 8250 controller (internal registers and behavior) for the guest BEFORE returning the real CPU back to the vCPU by executing the virtualization function.

> An example of this scenario - for an emulation that is not being used in our virtual machine example - is described in CVE-2019-6778 subtopic of QEMU CVEs session bellow

3.e-) QEMU devices acceleration

Virtio specification has been extensively described so far and you will find all information described at the **SR-IOV assignments: HW virtual functions**.

Idea of this session is to highlight the intersection between the Host OS and the Guest OS, where a security vulnerability could probably raise: the virtio-devices security surface.

From the same virtual machine XML description file we've been using so far, we can isolate the non-disk VIRTIO devices:

```
  <devices>
   <memballoon model='virtio'>
    <stats period='30'/>
    <alias name='balloon0'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x07' function='0x0'/>
    </memballoon>
    <rng model='virtio'>
      <rate bytes='1234' period='2000'/>
      <backend model='random'>/dev/urandom</backend>
      <alias name='rng0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x08' function='0x0'/>
    </rng>
  </devices>
```

> There is a specific VIRTIO disk session next (not being considered now)

And there aren't many para-paravirtualized devices configured (other than disks):

- memballoon: allows virtual machine memory to be reclaimed by the Host OS in order to better control its resources.

- random number generator: fills guest with entropy to its entropy pools.

Attack surface for this session is either one device or another:

1. *SECURITY*: **virtio rng**

Most of the security CVEs related to entropy is due to insufficient randomness (entropy) in a particular device that should act as a generator. QEMU implements some different rng implementations internally.

By using a VIRTIO rng device, GUEST requests entropy from QEMU process through the VIRTIO ring data structures, just like explained at session **How does VIRTIO devices work ?**. By default, QEMU provides entropy to a VIRTIO rng device by reading /dev/urandom of the Host OS (it could also be through gathering daemon: egd).

The HostOS kernel might be able to get entropy from different HW (or software) sources, most commonly:

- Intel RDRAND CPU instruction (pseudo-random number generator).
- Intel RDSEED instruction (better entropy based on thermal events).

This attack surface has **MUCH TO DO** with the side channel attack techniques called **Microarchitectural Data Sampling**, described in item **QEMU vCPU assignment & side-channel attacks** previously, but they are tightly related to a recent side-channel type vulnerability:

- Special Register Buffer Data Sampling

On affected processors, Intel has released microcode updates whose default behavior is to modify the RDRAND, RDSEED, and EGETKEY instructions to overwrite stale special register data in the shared staging buffer before the stale data can be accessed by any other logical processor on the same core or on a different core.

During execution of the RDRAND, RDSEED, or EGETKEY instructions, off-core accesses from other logical processors will be delayed until the special register read is complete and the stale data in the shared staging buffer is overwritten.

> Obviously this mitigation has performance impacts for overall RAND/SEED CPU instructions

2. *SECURITY*: **memory ballooning**

Attack surface here would be related to how the guest inflates or deflates its memory to force the HostOS to either provide it more memory, or reclaim memory being used by it. There is no automatic ballooning being done but there is an interface configured (device) to make it available if requested by the VMM administrator.

> CVE-2021-28039 is an example of ballooning CVE (although is for Xen hypervisor) trying to harm HostOS by exploring the ballooning configuration interface (in between hypervisor and guest).

3.f-) QEMU virtual disk devices

In previous item we've seen all accelerated (or sometimes called virtio) devices other than disks. Now it is time to concentrate a bit in the security surface of disks and how they can be accessed by virtual machines.

With our initial virtual machine definition example:

```
    <devices>
      <disk type='file' device='disk'>
        <driver name='QEMU' type='qcow2' cache='writethrough' error_policy='report'/>
        <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.qcow2' index='3'/>
        <backingStore type='file' index='4'>
          <format type='qcow2'/>
          <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.qcow2'/>
        </backingStore>
        <target dev='vda' bus='virtio'/>
        <alias name='virtio-disk0'/>
      </disk>
      <disk type='file' device='disk'>
        <driver name='QEMU' type='raw' cache='writethrough' error_policy='report'/>
        <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/init-disk' index='2'/>
        <backingStore/>
        <target dev='vdb' bus='virtio'/>
        <alias name='virtio-disk1'/>
      </disk>
      <disk type='file' device='disk'>
        <driver name='QEMU' type='raw' cache='writethrough' error_policy='report'/>
        <source file='xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx/swap' index='1'/>
        <backingStore/>
        <target dev='vdc' bus='virtio'/>
        <alias name='virtio-disk2'/>
      </disk>
    </devices>
```

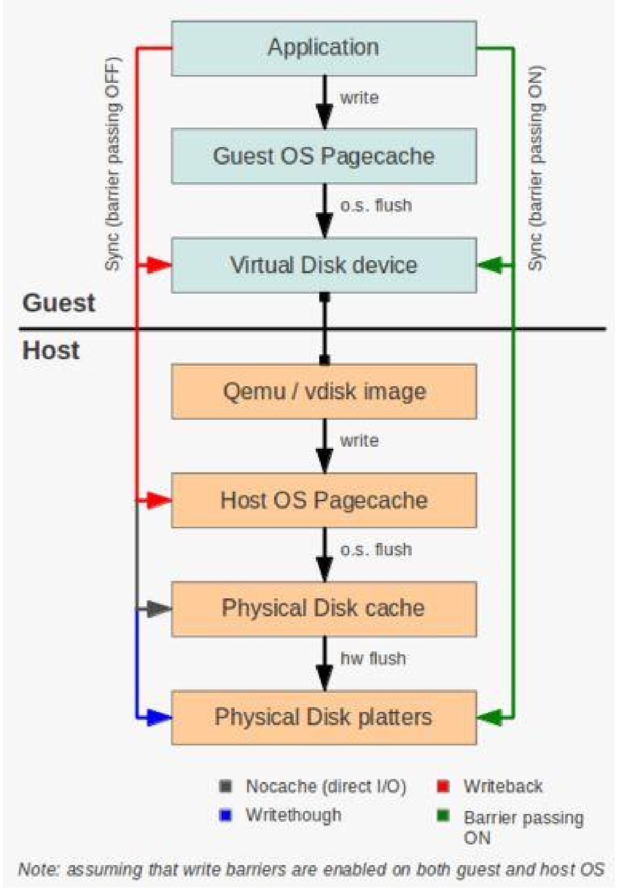
we have the following vdisks:

- virtio-disk0: vda virtio-blk device with a qcow2 backing device (root disk)
- virtio-disk1: vdb virtio-blk device with a raw backing device (swap disk)
- virtio-disk2: vdc virtio-blk device with a raw backing device (cloud-init)

As described earlier, there are many, many, ways to either emulate or accelerate a disk device, just like a network device. In this particular session we're only focusing in the virtio-blk approach (not virtio-scsi with other backing device/library). Thus, the security surface being described here is only related to virtio-blk devices (and not complete).

If you remember how the VIRTIO transport layer worked, with its vring buffers shared between the virtual machine and the QEMU process (or the kernel vhost thread on behalf of a virtual machine), you will also remember that this 'buffer intersection' is just one attack surface that exists when it comes to vdisks subject.

By also checking how vdisks can be cached by the HostOS:



It is clear that vdisk's security depends on the type of I/O being used by the virtual machine (sync, async, buffered, direct) and the HostOS pagecache exposure as well.

Historically, the virtio-blk driver/interface has always been very stable, with almost no CVEs. There were some old CVEs related to HEAP memory overflows because of bad memory alignment: CVE-2011-1750, but nothing recent.

> virtio-blk devices security surface depends on backing storage driver (Ceph, Gluster, Local File) internals.

> virtio-scsi devices security surface depends on backing storage driver (QEMU, LIO and libISCSI targets) internals.

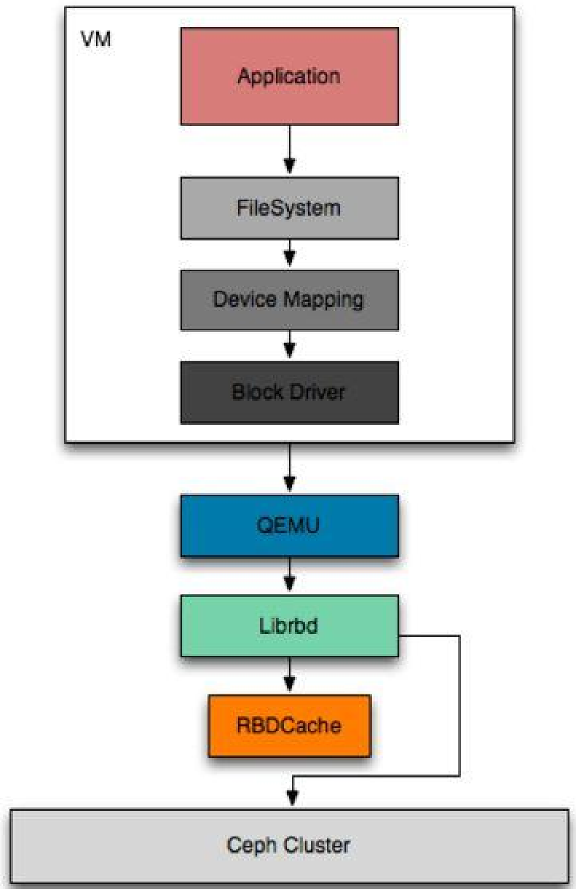
3.g-) QEMU storage backends

QEMU can use different storage backing technologies to support the virtual machine virtio-block device. Just like showed in the virtio-scsi case, in session **Block Devices, LUNs and Disks**, virtio-blk devices can be backed by technologies such as Ceph or Gluster.

Talking specifically about Ceph, Ceph's software libraries provide client applications with direct access to the reliable autonomic distributed object store (RADOS) object-based storage system, and also provide a foundation for some of Ceph's features, including RADOS Block Device (RBD), RADOS Gateway, and the Ceph File System.

Ceph's object storage system allows users to mount Ceph as a thin-provisioned block device. When an application writes data to Ceph using a block device, Ceph automatically stripes and replicates the data across the cluster. Ceph's RADOS Block Device (RBD) also integrates with Kernel-based Virtual Machines (KVMs).

This is how QEMU uses librbd as a storage backend engine:



and, obviously, each different backing storage mechanism increases or decreases the security surface depending on the resources that are shared with the HostOS and the mechanisms needed for it to operate (such as authentication, communication, and so).

3.h-) QEMU vNIC AND SR-IOV

Most of the network possible scenarios have been explained in the two previous sessions:

- **Device Para-Virtualization**: explaining VIRTIO acceleration
- **QEMU Internals**: explaining Virtio Implementation and Protocol

As described earlier, there are many different ways you can have a vNIC declared in a virtual machine and backed by either the QEMU process or the Host OS kernel (or some other off load user-land tool such as DPDK and/or OpenVswitch):

- virtio-net <=> tun/tap device (driven by QEMU)
- virtio-net <=> vhost-net (driven by kernel)
- virtio-net <=> openvswitch
- virtio-net <=> SR-IOV (backed by macvtap/vhost-user)
- virtio-net <=> SR-IOV (with VF directly assigned to guest)

As in other parts, this document is narrowing the scope to a specific virtual machine declaration and that is where we get the following vNIC from:

```
<interface type='direct'>
  <mac address='aa:bb:cc:aa:bb:cc'/>
  <source dev='if0aabbccaabbc' mode='passthrough'/>
  <target dev='macvtap0'/>
  <model type='virtio'/>
  <driver queues='3'/>
  <alias name='net0'/>
</interface>
```

As you can see this is not a typical VIRTIO <=> tun/tap scenario, nor a virtio-net <=> vhost-net typical scenario. This is a virtio-net <=> SR-IOV configuration in what we call 'indirect mode' (where the host OS kernel still has some role in packet processing, with macvtap driver). The SR-IOV configuration can be done in different ways:

- pass-through device access as SR-IOV VF PCI device (direct mode)
- use a pool of virtual function devices (direct mode)
- use a macvtap as the SR-IOV network adapter (indirect mode)

The indirect mode approach - of using a macvtap driver in the Host, letting the HostOS kernel to manage the virtual function network interfaces - makes the SR-IOV easier to be configured to virtual machines and allows live migrations to happen: you don't have to rely on PCI bus information to spin up a VM, you can simply tell the HostOS to use a free virtual function it already has configured.

Unfortunately the indirect mode adds some burden to the HostOS kernel, as it will have vhost kernel-threads coordinating data - shared memory - between the vm's VIRTIO buffers and the kernel threads, AND the macvtap driver responsible to transfer packets in and out to HW virtual functions.

This can also be seen by, like explained before, checking the existence of specific kernel threads in charge of the Vhost part of the interface for the QEMU process of the virtual machine in question:

```
1     0 60564     2  20  0      0     0 vhost_ S    ?        0:01 [vhost-60547]
1     0 60565     2  20  0      0     0 vhost_ S    ?        0:01 [vhost-60547]
1     0 60566     2  20  0      0     0 vhost_ S    ?        0:01 [vhost-60547]
```

By choosing different methods to deliver the SR-IOV virtual function to the guest you are also growing, or shrinking, the security surface of the system: If, instead of using macvtap, the SR-IOV virtual function was bound directly to the virtual machine, by using Intel VTd and IOMMU virtualization, virtio-net <=> vhost-net communication would be happening among the guest vm kernel memory and HW internal buffers only.

> With the macvtap approach, showed here, the HostOS kernel also has tasks manipulating/coordinating VIRTIO memory buffers (on behalf of the virtual machine).

3.i-) QEMU monitor (QMP and HMP)

There is a communication channel used for an external tool - like libvirt - to communicate with QEMU and operate its instances from external world:

- The QEMU Machine Protocol (QMP).

QMP is JSON based and features the following:

- Lightweight, text-based, easy to parse data format
- Asynchronous messages support (events)
- Capabilities negotiation
- API/ABI stability guarantees

Through QMP (or its Human read-able version: HMP) one can communicate with a QEMU instance and do many different tasks. A simple example checking the current status of a virtual machine is:

```
C: { "execute": "query-status" }
S: {
     "return": {
        "status": "running",
        "singlestep": false,
        "running": true
     }
   }
```

You can get all supported commands through the following command:

```
$ virsh qemu-monitor-command --domain <vm.name> --hmp --cmd "help info"

info balloon  -- show balloon information
info block [-n] [-v] [device] -- show info of one block device or all block devices (-n: show named nodes; -v: show
info block-jobs  -- show progress of ongoing block device operations
info blockstats  -- show block device statistics
info capture  -- show capture information
info chardev  -- show the character devices
info cpus  -- show infos for each CPU
info cpustats  -- show CPU statistics
info dump  -- Display the latest dump status
info history  -- show the command line history
info hotpluggable-cpus  -- Show information about hotpluggable CPUs
info ioapic  -- show io apic state
info iothreads  -- show iothreads
info irq  -- show the interrupts statistics (if available)
info jit  -- show dynamic compiler info
info KVM  -- show KVM information
info lapic [apic-id] -- show local apic state (apic-id: local apic to read, default is which of current CPU)
info mem  -- show the active virtual memory mappings
info memdev  -- show memory backends
info memory-devices  -- show memory devices
info memory_size_summary  -- show the amount of initially allocated and present hotpluggable (if enabled) me
info mice  -- show which guest mouse is receiving events
info migrate  -- show migration status
info migrate_cache_size  -- show current migration xbzrle cache size
info migrate_capabilities  -- show current migration capabilities
info migrate_parameters  -- show current migration parameters
info mtree [-f][-d][-o] -- show memory tree (-f: dump flat view for address spaces;-d: dump dispatch tree, valid w
info name  -- show the current VM name
info network  -- show the network state
info numa  -- show NUMA information
info opcount  -- show dynamic compiler opcode counters
info pci  -- show PCI info
info pic  -- show PIC state
info profile  -- show profiling information
info qdm  -- show qdev device model list
info qom-tree [path] -- show QOM composition tree
info qtree  -- show device tree
info ramblock  -- Display system ramblock information
info rdma  -- show RDMA state
info registers [-a] -- show the cpu registers (-a: all - show register info for all cpus)
info rocker name -- Show rocker switch
info rocker-of-dpa-flows name [tbl_id] -- Show rocker OF-DPA flow tables
info rocker-of-dpa-groups name [type] -- Show rocker OF-DPA groups
info rocker-ports name -- Show rocker ports
info roms  -- show roms
info sev  -- show SEV information
info snapshots  -- show the currently saved VM snapshots
info spice  -- show the spice server status
info status  -- show the current VM status (running|paused)
info sync-profile [-m] [-n] [max] -- show synchronization profiling info, up to max entries (default: 10), sorted by
info tlb  -- show virtual to physical memory mappings
info tpm  -- show the TPM device
info trace-events [name] [vcpu] -- show available trace-events & their state (name: event name pattern; vcpu: v
info usb  -- show guest USB devices
info usbhost  -- show host USB devices
info usernet  -- show user network stack connection states
info uuid  -- show the current VM UUID
info version  -- show the version of QEMU
info vm-generation-id  -- Show Virtual Machine Generation ID
info vnc  -- show the vnc server status
```

This command 'talks' in HMP format with the QEMU monitor device, giving it commands, or querying status of specific internal structures/emulations. Another example would be:

```
$ virsh qemu-monitor-command --domain <vm.name> --hmp --cmd info block -v

libvirt-1-format: /var/lib/libvirt/images/vm.name-disk01.qcow2 (qcow2)
  Attached to:     /machine/peripheral/virtio-disk0/virtio-backend
  Cache mode:      writeback

Images:
image: /var/lib/libvirt/images/vm.name-disk01.qcow2
file format: qcow2
virtual size: 30 GiB (32212254720 bytes)
disk size: 19.3 GiB
cluster_size: 65536
Format specific information:
  compat: 0.10
  refcount bits: 16
```

to get information from a particular VM block device backend.

As you might have already guessed, this is an important feature to be secured as it allows anyone with permissions to read/write to this interface to change practically anything in the virtual machine.

It is so important that one can get all internal - to the virtual machine - virtual address space of drivers mappings:

```
$ virsh qemu-monitor-command --domain <vm.name> --hmp --cmd info mtree -f -o

FlatView #0
 AS "memory", root: system
 AS "cpu-memory-0", root: system
 AS "cpu-memory-1", root: system
 AS "cpu-memory-2", root: system
 AS "cpu-memory-3", root: system
 AS "cpu-memory-4", root: system
 AS "cpu-memory-5", root: system
 AS "cpu-memory-6", root: system
 AS "cpu-memory-7", root: system
 AS "piix3-ide", root: bus master container
 AS "piix3-usb-uhci", root: bus master container
 AS "virtio-serial-pci", root: bus master container
 AS "virtio-blk-pci", root: bus master container
 AS "virtio-net-pci", root: bus master container
 AS "virtio-balloon-pci", root: bus master container
 Root memory region: system
  0000000000000000-00000000000bffff (prio 0, ram): pc.ram owner:{obj path=/objects/pc.ram} KVM
  00000000000c0000-00000000000c0fff (prio 0, rom): pc.ram @00000000000c0000 owner:{obj path=/objects/p
  00000000000c1000-00000000000c3fff (prio 0, ram): pc.ram @00000000000c1000 owner:{obj path=/objects/p
  00000000000c4000-00000000000e7fff (prio 0, rom): pc.ram @00000000000c4000 owner:{obj path=/objects/p
  00000000000e8000-00000000000effff (prio 0, ram): pc.ram @00000000000e8000 owner:{obj path=/objects/p
  00000000000f0000-00000000000fffff (prio 0, rom): pc.ram @00000000000f0000 owner:{obj path=/objects/pc.
  0000000000100000-00000000bfffffff (prio 0, ram): pc.ram @0000000000100000 owner:{obj path=/objects/pc.
  00000000feb80000-00000000feb8002f (prio 0, i/o): msix-table owner:{dev id=net0}
  00000000feb80800-00000000feb80807 (prio 0, i/o): msix-pba owner:{dev id=net0}
  00000000feb81000-00000000feb8101f (prio 0, i/o): msix-table owner:{dev id=virtio-serial0}
  00000000feb81800-00000000feb81807 (prio 0, i/o): msix-pba owner:{dev id=virtio-serial0}
```

```
    00000000feb82000-00000000feb8201f (prio 0, i/o): msix-table owner:{dev id=virtio-disk0}
    00000000feb82800-00000000feb82807 (prio 0, i/o): msix-pba owner:{dev id=virtio-disk0}
    00000000feb83000-00000000feb8300f (prio 1, i/o): i6300esb owner:{dev id=watchdog0}
    00000000febf0000-00000000febf0fff (prio 0, i/o): virtio-pci-common owner:{dev id=net0}
    00000000febf1000-00000000febf1fff (prio 0, i/o): virtio-pci-isr owner:{dev id=net0}
    00000000febf2000-00000000febf2fff (prio 0, i/o): virtio-pci-device owner:{dev id=net0}
    00000000febf3000-00000000febf3fff (prio 0, i/o): virtio-pci-notify owner:{dev id=net0}
    00000000febf4000-00000000febf4fff (prio 0, i/o): virtio-pci-common owner:{dev id=virtio-serial0}
    00000000febf5000-00000000febf5fff (prio 0, i/o): virtio-pci-isr owner:{dev id=virtio-serial0}
    00000000febf6000-00000000febf6fff (prio 0, i/o): virtio-pci-device owner:{dev id=virtio-serial0}
    00000000febf7000-00000000febf7fff (prio 0, i/o): virtio-pci-notify owner:{dev id=virtio-serial0}
    00000000febf8000-00000000febf8fff (prio 0, i/o): virtio-pci-common owner:{dev id=virtio-disk0}
    00000000febf9000-00000000febf9fff (prio 0, i/o): virtio-pci-isr owner:{dev id=virtio-disk0}
    00000000febfa000-00000000febfafff (prio 0, i/o): virtio-pci-device owner:{dev id=virtio-disk0}
    00000000febfb000-00000000febfbfff (prio 0, i/o): virtio-pci-notify owner:{dev id=virtio-disk0}
    00000000febfc000-00000000febfcfff (prio 0, i/o): virtio-pci-common owner:{dev id=balloon0}
    00000000febfd000-00000000febfdfff (prio 0, i/o): virtio-pci-isr owner:{dev id=balloon0}
    00000000febfe000-00000000febfefff (prio 0, i/o): virtio-pci-device owner:{dev id=balloon0}
    00000000febff000-00000000febfffff (prio 0, i/o): virtio-pci-notify owner:{dev id=balloon0}
    00000000fec00000-00000000fec00fff (prio 0, i/o): KVM-ioapic owner:{dev path=/machine/i440fx/ioapic}
    00000000fed00000-00000000fed003ff (prio 0, i/o): hpet owner:{dev path=/machine/unattached/device[14]}
```

Security surface related to this communication channel is either a file or a file descriptor, opened by either libvirt or QEMU with that intent. From the QEMU instance command line:

```
libvirt+    3660  0.0  1.6 6357284 3250448 ?     Sl   May19 144:53 /bin/QEMU-system-x86_64 -name guest=_dropb
```

we have:

```
chardev socket,id=charmonitor,fd=29,server,nowait -mon chardev=charmonitor,id=monitor,mode=control
```

which means libvirt has a file descriptor pointing to a unix socket that has likely being unlinked already - so it will vanish after the socket is closed - and it passed the file descriptor to QEMU process after exec'ing it.

If one is executing QEMU directly, it is possible to have different files/sockets opened to QEMU/HMP by having QEMU cmdline with the following arguments:

```
-chardev socket,id=monitor,path=/tmp/guest.monitor,server,nowait \
-monitor chardev:monitor \
-chardev socket,id=serial,path=/tmp/guest.serial,server,nowait \
-serial chardev:serial \
-qmp unix:/tmp/guest.sock,server,nowait \
```

and then using either screen (or minicom) to open the unix socket:

```
sudo minicom -D unix\#/tmp/kguest.monitor   # HMP (Human Monitor)   - txt
sudo minicom -D unix\#/tmp/kguest.socket    # QMP (Machine Monitor) - json
sudo minicom -D unix\#/tmp/kguest.serial    # Guest Serial Console
```

and communicate with the QEMU instance by using QMP/HMP commands.

> This session shows that, despite not being obvious, the QEMU instance security surface might be extended to many different things such as internal (to libvirt and QEMU) communication channels.

3.j-) QEMU confinement

As we've seen, QEMU supports many different use cases, some of which have stricter security requirements than others. The community has agreed on the overall security requirements that users may depend on. These requirements define what is considered supported from a security perspective.

3.j.1-) Guest isolation

Guest isolation is the confinement of guest code to the virtual machine. When guest code gains control of execution on the host this is called escaping the virtual machine. Isolation also includes resource limits such as throttling of CPU, memory, disk, or network. Guests must be unable to exceed their resource limits.

QEMU presents an attack surface to the guest in the form of emulated devices. The guest must not be able to gain control of QEMU. Bugs in emulated devices could allow malicious guests to gain code execution in QEMU. At this point the guest has escaped the virtual machine and is able to act in the context of the QEMU process on the host.

Guests often interact with other guests and share resources with them. A malicious guest must not gain control of other guests or access their data. Disk image files and network traffic must be protected from other guests unless explicitly shared between them by the user.

3.j.2-) Principle of Least Privilege

The principle of least privilege states that each component only has access to the privileges necessary for its function. In the case of QEMU this means that each process only has access to resources belonging to the guest.

The QEMU process should not have access to any resources that are inaccessible to the guest. This way the guest does not gain anything by escaping into the QEMU process since it already has access to those same resources from within the guest.

Following the principle of least privilege immediately fulfills guest isolation requirements. For example, guest A only has access to its own disk image file a.img and not guest B’s disk image file b.img.

In reality certain resources are inaccessible to the guest but must be available to QEMU to perform its function. For example, host system calls are necessary for QEMU but are not exposed to guests. A guest that escapes into the QEMU process can then begin invoking host system calls.

New features must be designed to follow the principle of least privilege. Should this not be possible for technical reasons, the security risk must be clearly documented so users are aware of the trade-off of enabling the feature.

3.j.3-) Isolation Mechanisms

Several isolation mechanisms are available to realize this architecture of guest isolation and the principle of least privilege. With the exception of Linux seccomp, these mechanisms are all deployed by management tools that launch QEMU, such as libvirt.

The fundamental isolation mechanism is that QEMU processes must run as unprivileged users. Sometimes it seems more convenient to launch QEMU as root to give it access to host devices (e.g. /dev/net/tun) but this poses a huge security risk. File descriptor passing can be used to give an otherwise unprivileged QEMU process access to host devices without running QEMU as root. It is also possible to launch QEMU as a non-root user and configure UNIX groups for access to /dev/KVM, /dev/net/tun, and other device nodes. Some Linux distros already ship with UNIX groups for these devices by default.

- SELinux and AppArmor make it possible to confine processes beyond the traditional UNIX process and file permissions model. They restrict the QEMU process from accessing processes and files on the host system that are not needed by QEMU.
- Resource limits and cgroup controllers provide throughput and utilization limits on key resources such as CPU time, memory, and I/O bandwidth.
- Linux namespaces can be used to make process, file system, and other system resources unavailable to QEMU. A namespaced QEMU process is restricted to only those resources that were granted to it.
- Linux seccomp is available via the QEMU --sandbox option. It disables system calls that are not needed by QEMU, thereby reducing the host kernel attack surface.

> IMPORTANT IDEA HERE IS that, by having an isolation mechanism, one escape due to a security vulnerability, for example, would also have to break the isolation mechanism in order to be fully prejudicial to other OS resources (most of the times).

Some important isolation mechanisms:

3.j.4-) Isolation: apparmor

AppArmor ("Application Armor") is a Linux kernel security module that allows the system administrator to restrict programs' capabilities with per-program profiles. Profiles can allow capabilities like network access, raw socket access, and the permission to read, write, or execute files on matching paths. AppArmor supplements the traditional Unix discretionary access control (DAC) model by providing mandatory access control (MAC).

First, the libvirtd process is considered trusted and is therefore confined with a lenient profile that allows the libvirt daemon to launch VMs or containers, change into another AppArmor profile and use virt-aa-helper to manipulate AppArmor profiles. virt-aa-helper is a helper application that can add, remove, modify, load and unload AppArmor profiles in a limited and restricted way. libvirtd is not allowed to adjust anything in /sys/kernel/security directly, or modify the profiles for the virtual machines directly. Instead, libvirtd must use virt-aa-helper, which is itself run under a very restrictive AppArmor profile. Using this architecture helps prevent any opportunities for a subverted libvirtd to change its own profile (especially useful if the libvirtd profile is adjusted to be restrictive) or modify other AppArmor profiles on the system.

libvirt apparmor important files:

- /etc/apparmor.d/usr.sbin.libvirtd : profile for libvirtd
- /etc/apparmor.d/usr.lib.virt-aa-helper : profile for virt-aa-helper
- /etc/apparmor.d/libvirt/TEMPLATE.QEMU : consulted to create new profile
- /etc/apparmor.d/abstractions/libvirt-qemu : shared among all VMs
- /etc/apparmor.d/libvirt/libvirt- unique base profile for a VM
- /etc/apparmor.d/libvirt/libvirt-.files guest-specific files required

The confinement process is as follows (assume the VM has a libvirt UUID of 'xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx'):

If libvirtd is both, confined by AppArmor and configured to use it in /etc/libvirt/QEMU.conf, it will use the AppArmor security driver. When a VM is started, libvirtd decides whether to ask virt-aa-helper to create a new profile or modify an existing one. If no profile exists, libvirtd asks virt-aa-helper to generate the new base profile, in this case /etc/apparmor.d/libvirt/libvirt-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx, which it does based on /etc/apparmor.d/libvirt/TEMPLATE.QEMU.

Notice, the new profile has a profile name that is based on the guest’s UUID. Once the base profile is created, virt-aa-helper works the same for create and modify: virt-aa-helper will determine what files are required for the guest to run (eg kernel, initrd, disk, serial, etc), updates /etc/apparmor.d/libvirt/libvirt-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx.files, then loads the profile into the kernel.

libvirtd will proceed as normal at this point, until just before it forks a QEMU/KVM/container process, it will call aa_change_profile() to transition into the profile ‘libvirt-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx’ (the one virt-aa-helper loaded into the kernel in the previous step).

When the VM is shutdown, libvirtd asks virt-aa-helper to remove the profile, and virt-aa-helper unloads the profile from the kernel.

From the examples being used in this document, you will see the process just described by executing:

```
$ ls -a1 /etc/apparmor.d/libvirt/

TEMPLATE.lxc
TEMPLATE.QEMU
libvirt-6814edd9-cd66-4256-a603-6f6591841c8c
libvirt-6814edd9-cd66-4256-a603-6f6591841c8c.files

$ sudo aa-status
```

```
apparmor module is loaded.
17 profiles are loaded.
17 profiles are in enforce mode.
   ...
   /sbin/capause
   /sbin/computeproxy
   /sbin/dhclient
   /usr/lib/ipsec/stroke
   /usr/sbin/swanctl
   /usr/sbin/tcpdump
   docker-default
   fluentd-logs
   fluentd-qradar
   libvirt-6814edd9-cd66-4256-a603-6f6591841c8c
   libvirtd
   libvirtd/QEMU_bridge_helper
   virt-aa-helper
0 profiles are in complain mode.
133 processes have profiles defined.
132 processes are in enforce mode.
   /sbin/computeproxy (12410)
   docker-default (722)
   ...
   docker-default (65128)
   libvirt-6814edd9-cd66-4256-a603-6f6591841c8c (60547)
   libvirtd (12355)
0 processes are in complain mode.
1 processes are unconfined but have a profile defined.
   /usr/lib/ipsec/charon (6245)
```

Considering all the new VM profiles will be based on:

- /etc/apparmor.d/libvirt/TEMPLATE.QEMU : consulted to create new profile

and seeing its contents:

```
#include <tunables/global>

profile LIBVIRT_TEMPLATE flags=(attach_disconnected) {
  #include <abstractions/libvirt-qemu>
}
```

```
$ cat /etc/apparmor.d/abstractions/libvirt-qemu
```

You will realize there is room for maneuvers for more or less confinement based on the type of network and storage backends you're using to support the virtio-net / virtio-blk and/or virtio-xxx devices you're delivering to the virtual machines.

```
  #include <abstractions/base>
  #include <abstractions/consoles>
  #include <abstractions/nameservice>

  # required for reading disk images
  capability dac_override,
  capability dac_read_search,
  capability chown,
  # needed to drop privileges
  capability setgid,
  capability setuid,

  # for 9p
  capability fsetid,
  capability fowner,

  network inet stream,
  network inet6 stream,

  ptrace (readby, tracedby) peer=libvirtd,
  ptrace (readby, tracedby) peer=/usr/sbin/libvirtd,

  signal (receive) peer=libvirtd,
  signal (receive) peer=/usr/sbin/libvirtd,

  /dev/KVM rw,
  /dev/net/tun rw,
  /dev/ptmx rw,
  @{PROC}/*/status r,
  # When QEMU is signaled to terminate, it will read cmdline of signaling
  # process for reporting purposes. Allowing read access to a process
  # cmdline may leak sensitive information embedded in the cmdline.
  @{PROC}/@{pid}/cmdline r,
  # Per man(5) proc, the kernel enforces that a thread may
  # only modify its comm value or those in its thread group.
  owner @{PROC}/@{pid}/task/@{tid}/comm rw,
  @{PROC}/sys/kernel/cap_last_cap r,
  owner @{PROC}/*/auxv r,
  @{PROC}/sys/vm/overcommit_memory r,

  # For hostdev access. The actual devices will be added dynamically
  /sys/bus/usb/devices/ r,
  /sys/devices/**/usb[0-9]*/** r,
  # libusb needs udev data about usb devices (~equal to content of lsusb -v)
  /run/udev/data/+usb* r,
  /run/udev/data/c16[6,7]* r,
  /run/udev/data/c18[0,8,9]* r,

  # WARNING: this gives the guest direct access to host hardware and specific
  # portions of shared memory. This is required for sound using ALSA with KVM,
  # but may constitute a security risk. If your environment does not require
  # the use of sound in your VMs, feel free to comment out or prepend 'deny' to
  # the rules for files in /dev.
  /dev/snd/* rw,
  /{dev,run}/shm r,
  /{dev,run}/shmpulse-shm* r,
  /{dev,run}/shmpulse-shm* rwk,
  capability ipc_lock,
  # spice
  owner /{dev,run}/shm/spice.* rw,
  # 'kill' is not required for sound and is a security risk. Do not enable
  # unless you absolutely need it.
  deny capability kill,

  # Uncomment the following if you need access to /dev/fb*
  #/dev/fb* rw,

  /etc/pulse/client.conf r,
  @{HOME}/.pulse-cookie rwk,
  owner /root/.pulse-cookie rwk,
  owner /root/.pulse/ rw,
  owner /root/.pulse/* rw,
  /usr/share/alsa/** r,
  owner /tmp/pulse-*/ rw,
  owner /tmp/pulse-*/* rw,
  /var/lib/dbus/machine-id r,

  # access to firmware's etc
  /usr/share/AAVMF/** r,
  /usr/share/bochs/** r,
  /usr/share/edk2-ovmf/** r,
  /usr/share/KVM/** r,
  /usr/share/misc/sgabios.bin r,
  /usr/share/openbios/** r,
  /usr/share/openhackware/** r,
  /usr/share/OVMF/** r,
  /usr/share/ovmf/** r,
  /usr/share/proll/** r,
  /usr/share/QEMU-efi/** r,
  /usr/share/QEMU-KVM/** r,
  /usr/share/QEMU/** r,
  /usr/share/seabios/** r,
  /usr/share/sgabios/** r,
  /usr/share/slof/** r,
  /usr/share/vgabios/** r,

  # pki for libvirt-vnc and libvirt-spice (LP: #901272, #1690140)
  /etc/pki/CA/ r,
  /etc/pki/CA/* r,
  /etc/pki/libvirt{,-spice,-vnc}/ r,
  /etc/pki/libvirt{,-spice,-vnc}/** r,
  /etc/pki/QEMU/ r,
  /etc/pki/QEMU/** r,

  # the various binaries
  /usr/bin/KVM rmix,
  /usr/bin/qemu rmix,
  /usr/bin/qemu-aarch64 rmix,
  /usr/bin/qemu-alpha rmix,
  /usr/bin/qemu-arm rmix,
  /usr/bin/qemu-armeb rmix,
  /usr/bin/qemu-cris rmix,
  /usr/bin/qemu-i386 rmix,
  /usr/bin/qemu-KVM rmix,
  /usr/bin/qemu-m68k rmix,
  /usr/bin/qemu-microblaze rmix,
  /usr/bin/qemu-microblazeel rmix,
  /usr/bin/qemu-mips rmix,
  /usr/bin/qemu-mips64 rmix,
  /usr/bin/qemu-mips64el rmix,
  /usr/bin/qemu-mipsel rmix,
  /usr/bin/qemu-mipsn32 rmix,
  /usr/bin/qemu-mipsn32el rmix,
  /usr/bin/qemu-or32 rmix,
  /usr/bin/qemu-ppc rmix,
  /usr/bin/qemu-ppc64 rmix,
  /usr/bin/qemu-ppc64abi32 rmix,
  /usr/bin/qemu-ppc64le rmix,
  /usr/bin/qemu-s390x rmix,
  /usr/bin/qemu-sh4 rmix,
  /usr/bin/qemu-sh4eb rmix,
  /usr/bin/qemu-sparc rmix,
  /usr/bin/qemu-sparc32plus rmix,
  /usr/bin/qemu-sparc64 rmix,
  /usr/bin/qemu-system-aarch64 rmix,
  /usr/bin/qemu-system-alpha rmix,
  /usr/bin/qemu-system-arm rmix,
  /usr/bin/qemu-system-cris rmix,
  /usr/bin/qemu-system-hppa rmix,
  /usr/bin/qemu-system-i386 rmix,
  /usr/bin/qemu-system-lm32 rmix,
  /usr/bin/qemu-system-m68k rmix,
  /usr/bin/qemu-system-microblaze rmix,
  /usr/bin/qemu-system-microblazeel rmix,
  /usr/bin/qemu-system-mips rmix,
  /usr/bin/qemu-system-mips64 rmix,
  /usr/bin/qemu-system-mips64el rmix,
  /usr/bin/qemu-system-mipsel rmix,
  /usr/bin/qemu-system-moxie rmix,
  /usr/bin/qemu-system-nios2 rmix,
  /usr/bin/qemu-system-or1k rmix,
  /usr/bin/qemu-system-or32 rmix,
  /usr/bin/qemu-system-ppc rmix,
  /usr/bin/qemu-system-ppc64 rmix,
  /usr/bin/qemu-system-ppcemb rmix,
  /usr/bin/qemu-system-riscv32 rmix,
  /usr/bin/qemu-system-riscv64 rmix,
  /usr/bin/qemu-system-s390x rmix,
  /usr/bin/qemu-system-sh4 rmix,
  /usr/bin/qemu-system-sh4eb rmix,
  /usr/bin/qemu-system-sparc rmix,
  /usr/bin/qemu-system-sparc64 rmix,
  /usr/bin/qemu-system-tricore rmix,
  /usr/bin/qemu-system-unicore32 rmix,
  /usr/bin/qemu-system-x86_64 rmix,
  /usr/bin/qemu-system-xtensa rmix,
  /usr/bin/qemu-system-xtensaeb rmix,
  /usr/bin/qemu-unicore32 rmix,
  /usr/bin/qemu-x86_64 rmix,
  # for Debian/Ubuntu QEMU-block-extra / RPMs QEMU-block-* (LP: #1554761)
  /usr/{lib,lib64}/QEMU/*.so mr,
  /usr/lib/@{multiarch}/QEMU/*.so mr,

  # let QEMU load old shared objects after upgrades (LP: #1847361)
  /{var/,}run/QEMU/*/*.so mr,
  # but explicitly deny with auditing writing to these files
  audit deny /{var/,}run/QEMU/*/*.so w,

  # swtpm
  /{usr/,}bin/swtpm rmix,
  /usr/{lib,lib64}/libswtpm_libtpms.so mr,
  /usr/lib/@{multiarch}/libswtpm_libtpms.so mr,

  # for save and resume
  /{usr/,}bin/dash rmix,
  /{usr/,}bin/dd rmix,
  /{usr/,}bin/cat rmix,
  # for restore
  /{usr/,}bin/bash rmix,

  # for usb access
  /dev/bus/usb/ r,
  /etc/udev/udev.conf r,
  /sys/bus/ r,
  /sys/class/ r,

  # for rbd
  /etc/ceph/ceph.conf r,

  # Various functions will need to enumerate /tmp (e.g. ceph), allow the base
  # dir and a few known functions like samba support.
  # We want to avoid to give blanket rw permission to everything under /tmp,
  # users are expected to add site specific addons for more uncommon cases.
  # QEMU processes usually all run as the same users, so the "owner"
  # restriction prevents access to other services files, but not across
  # different instances.
  # This is a tradeoff between usability and security - if paths would be more
  # predictable that would be preferred - at least for write rules we would
  # want more unique paths per rule.
  /{,var/}tmp/ r,
  owner /{,var/}tmp/**/ r,

  # for file-posix getting limits since 9103f1ce
  /sys/devices/**/block/*/queue/max_segments r,

  # for ppc device-tree access
  @{PROC}/device-tree/ r,
  @{PROC}/device-tree/** r,
  /sys/firmware/devicetree/** r,

  # allow connect with openGraphicsFD to work
  unix (send, receive) type=stream addr=none peer=(label=libvirtd),
  unix (send, receive) type=stream addr=none peer=(label=/usr/sbin/libvirtd),

  # allow access to charm-specific ceph config (LP: #1403648).
  # No more silencing spurious denials as it can more critically hide other issues (LP: #1719579)
  # Also allow the optional asok key that might be enabled by the charm (LP: #1779674)
  /var/lib/charm/*/ceph.conf r,
  /run/ceph/rbd-client-*.asok rw,

  # KVM.powerpc executes/accesses this
  /{usr/,}bin/uname rmix,
  /{usr/,}sbin/ppc64_cpu rmix,
  /{usr/,}bin/grep rmix,
  /sys/devices/system/cpu/subcores_per_core r,
  /sys/devices/system/cpu/cpu*/online r,

  # for gathering information about available host resources
  /sys/devices/system/cpu/ r,
  /sys/devices/system/node/ r,
  /sys/devices/system/node/node[0-9]*/meminfo r,
  /sys/module/vhost/parameters/max_mem_regions r,

  # silence refusals to open lttng files (see LP: #1432644)
  deny /dev/shm/lttng-ust-wait-* r,
  deny /run/shm/lttng-ust-wait-* r,
  # for vfio hotplug on systems without static vfio (LP: #1775777)
  /dev/vfio/vfio rw,

  # for vhost-net/vsock/scsi hotplug (LP: #1815910)
  /dev/vhost-net rw,
  /dev/vhost-vsock rw,
  /dev/vhost-scsi rw,

  # required for sasl GSSAPI plugin
  /etc/gss/mech.d/ r,
  /etc/gss/mech.d/* r,

  # required by libpmem init to fts_open()/fts_read() the symlinks in
  # /sys/bus/nd/devices
  / r, # harmless on any lsb compliant system
  /sys/bus/nd/devices/{,**/} r,

  # Site-specific additions and overrides. See local/README for details.
  #include <local/abstractions/libvirt-qemu>
```

Just ONE quick example, that **could** be discussed here - in order to minimize chances of an escaped QEMU process to execute something else in the Host OS, could be:

```
# for save and resume
/{usr/,}bin/dash rmix,
/{usr/,}bin/dd rmix,
/{usr/,}bin/cat rmix,

# for restore
/{usr/,}bin/bash rmix,
```

rules allowing QEMU process to execute shells (a typical security scenario where bytecodes executing /bin/bash are executed because of memory addressing exploits). Many other mitigation scenarios could be discussed here, making sure the TEMPLATE for any new QEMU profile.

@ - PROPOSITION 1

> **THIS IS THE FIRST PROPOSITION OF THIS DOCUMENT**
>
> It is virtually impossible to cover all attack surface of QEMU and its para-virtualization techniques (shown in this document exhaustively). by **cover** we mean **predict and remediate**. >
>
> Instead of logging/introspecting erratic behaviors, that **could** indicate a security breakage attempt, it is better to concentrate efforts in isolating QEMU resources so, if a escape ever happens, it wouldn't be able to deeply affect other resources.
>
> Some observations:
>
> 1. Yes, this is already implemented through existing apparmor profiles/templates. We would have to review the 1:1 mapping of each needed resource and the existing profiles, discarding rules that aren't needed and making sure all needed rules exist. Summary: to get rid of default apparmor profiles/templates and create a specific set.
>
> 2. All access pattern deviations from the existing QEMU infrastructure would have to be considered when developing a new features. One example could be... changing the virtio-blk backing device from a QCOW2 file, from a libvirt disk pool backend, to making it a RBD device, through the librbd storage backend.
>
> Summary: profiles would have to be changed according to new features.
>
> 3. Some of the attack surface won't be covered by that and should, continue, be mitigated by other techniques, specially all the HW side-channel attack techniques: regarding vCPUs placement, cache

> invalidation/observation and such.

3.j.5-) Isolation: cgroups (as resources manager)

The QEMU and LXC drivers make use of the Linux "Control Groups" facility for applying resource management to their virtual machines and containers.

- Required controllers

The control groups filesystem supports multiple "controllers". By default systemd should mount all controllers compiled into the kernel at /sys/fs/cgroup/*controller*. Libvirt will never attempt to mount any controllers itself, merely detect where they are mounted.

The QEMU driver is capable of using the **cpuset, cpu, cpuacct, memory, blkio and devices controllers**. None of them are compulsory. If any controller is NOT MOUNTED, the resource management APIs which use it will CEASE TO OPERATE.

- **Current cgroups layout** The layout is based on the concepts of **partitions** and **consumers**. A "consumer" is a cgroup which holds the processes for a single virtual machine. A "partition" is a cgroup which does not contain any processes, but can have resource controls applied. A "partition" will have zero or more child directories which may be either "consumer" or "partition".
- **Systemd cgroups integration** Each consumer maps to a systemd scope unit, while partitions map to a system slice unit.
- **Systemd scope naming** The systemd convention is for the scope name of virtual machines to be of the general format machine-$NAME.scope. Libvirt forms the $NAME part of this by concatenating the driver type with the id and truncated name of the guest, and then escaping any systemd reserved characters.
- **Systemd slice naming** systemd convention for slice naming is that a slice should include the name of all of its parents prepended on its own name. So for a libvirt partition /machine/engineering/testing, the slice name will be machine-engineering-testing.slice. The slice names map directly to the cgroup directory names. Systemd creates three top level slices by default, system.slice user.slice and machine.slice. All virtual machines or containers created by libvirt will be associated with machine.slice by default.
- **Systemd cgroup layout example**

```
$ROOT
  |
  +- system.slice
  |   |
  |   +- libvirtd.service
  |
  +- machine.slice
      |
      +- machine-QEMU\x2d1\x2dvm1.scope
      |   |
      |   +- libvirt
      |       |
      |       +- emulator
      |       +- vcpu0
      |       +- vcpu1
      |
      +- machine-QEMU\x2d2\x2dvm2.scope
      |   |
      |   +- libvirt
      |       |
      |       +- emulator
      |       +- vcpu0
      |       +- vcpu1
      |
      +- machine-QEMU\x2d3\x2dvm3.scope
```

```
|   |
|   +- libvirt
|       |
|       +- emulator
|       +- vcpu0
|       +- vcpu1
|
+- machine-engineering.slice
|   |
|   +- machine-engineering-testing.slice
|   |   |
|   |   +- machine-lxc\x2d11111\x2dcontainer1.scope
|   |
|   +- machine-engineering-production.slice
|       |
|       +- machine-lxc\x2d22222\x2dcontainer2.scope
|
+- machine-marketing.slice
    |
    +- machine-lxc\x2d33333\x2dcontainer3.scope
```

- **Using custom partitions** A single default partition /machine setup may not be sufficiently flexible to apply resource constraints to groups of VMs. The administrator may wish to sub-divide the default partition, for example into "testing" and "production" partitions, and then assign each guest to a specific sub-partition. This is achieved via a small element addition to the guest domain XML config, just below the main domain element

```xml
<resource>
  <partition>/machine/production</partition>
</resource>
```

3.j.6-) Isolation: namespaces

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication.

namespace kinds

Namespace functionality is the same across all kinds: each process is associated with a namespace and can only see or use the resources associated with that namespace, and descendant namespaces where applicable. This way each process (or process group thereof) can have a unique view on the resources. Which resource is isolated depends on the kind of namespace that has been created for a given process group.

> Namespace feature can either be seen as a virtualization OR isolation feature.

- Mount (mnt)

Mount namespaces control mount points. Upon creation the mounts from the current mount namespace are copied to the new namespace, but mount points created afterwards do not propagate between namespaces (using shared subtrees, it is possible to propagate mount points between namespaces).

- **Process ID (pid)**

The PID namespace provides processes with an independent set of process IDs (PIDs) from other namespaces. PID namespaces are nested, meaning when a new process is created it will have a PID for each namespace from its current namespace up to the initial PID namespace. Hence the initial PID namespace is able to see all processes, albeit with different PIDs than other namespaces will see processes with.

The first process created in a PID namespace is assigned the process id number 1 and receives most of the same special treatment as the normal init process, most notably that orphaned processes within the namespace are attached to it. This also means that the termination of this PID 1 process will immediately terminate all processes in its PID namespace and any descendants.

- Network (net)

Network namespaces virtualize the network stack. On creation a network namespace contains only a loopback interface.

Each network interface (physical or virtual) is present in exactly 1 namespace and can be moved between namespaces.

Each namespace will have a private set of IP addresses, its own routing table, socket listing, connection tracking table, firewall, and other network-related resources.

Destroying a network namespace destroys any virtual interfaces within it and moves any physical interfaces within it back to the initial network namespace.

- Interprocess Communication (ipc)

IPC namespaces isolate processes from SysV style inter-process communication. This prevents processes in different IPC namespaces from using, for example, the SHM family of functions to establish a range of shared memory between the two processes. Instead each process will be able to use the same identifiers for a shared memory region and produce two such distinct regions.

- UTS

UTS (UNIX Time-Sharing) namespaces allow a single system to appear to have different host and domain names to different processes.

- **User ID (user)**

User namespaces are a feature to provide both privilege isolation and user identification segregation across multiple sets of processes available.

With administrative assistance it is possible to build a container with seeming administrative rights without actually giving elevated privileges to user processes. Like the PID namespace, user namespaces are nested and each new user namespace is considered to be a child of the user namespace that created it.

A user namespace contains a mapping table converting user IDs from the container's point of view to the system's point of view. This allows, for example, the root user to have user id 0 in the container but is actually treated as user id 1,400,000 by the system for ownership checks. A similar table is used for group id mappings and ownership checks.

To facilitate privilege isolation of administrative actions, each namespace type is considered owned by a user namespace based on the active user namespace at the moment of creation. A user with administrative privileges in the appropriate user namespace will be allowed to perform administrative actions within that other namespace type. For example, if a process has administrative permission to change the IP address of a network interface, it may do so as long as its own user namespace is the same as (or ancestor of) the user namespace that owns the network namespace. Hence the initial user namespace has administrative control over all namespace types in the system.

- Control group (cgroup) Namespace

The cgroup namespace type hides the identity of the control group of which process is a member. A process in such a namespace, checking which control group any process is part of, would see a path that is actually relative to the control group set at creation time, hiding its true control group position and identity.

- Time Namespace

The time namespace allows processes to see different system times in a way similar to the UTS namespace.

@ - PROPOSITION 2

> THIS IS THE SECOND PROPOSITION OF THIS DOCUMENT STEP

> To confine QEMU processes in different namespaces (or even containers) in order to guarantee OS level isolation to different virtual machine tenants as libvirt runs all QEMU processes with the same OS user 'libvirt-qemu'.
>
> In case there is a escape from QEMU, from any security surface described in this document, confining the processes **at least** in different two namespaces: **process id** and **user id**, might mitigate all other virtual machines being compromised.
>
> OBS: You can see this is the tendency in QEMU as virtio-fs, a relatively new feature for sharing filesystems among the Host and the Guests, has sandbox support and it does use different PID, MOUNT and NETWORK namespaces for the feature.

3.j.7-) Isolation: seccomp (sandbox)

seccomp (short for secure computing mode): is a computer security facility in the Linux kernel. seccomp allows a process to make a one-way transition into a "secure" state where it cannot make any system calls except exit(), sigreturn(), read() and write() to already-open file descriptors. Should it attempt any other system calls, the kernel will terminate the process with SIGKILL or SIGSYS.In this sense, it does not virtualize the system's resources but ISOLATES the process from them entirely.

QEMU seccomp

Linux seccomp is available, and being used in the example being used for this document, via the QEMU --sandbox option. It disables system calls that are not needed by QEMU, reducing the host kernel attack surface.

Our example has the following cmdline argument in QEMU:

```
-sandbox on,obsolete=deny,elevateprivileges=deny,spawn=deny,resourcecontrol=deny
```

- on: enables seccomp for filtering system calls mode
- elevatedprivileges: disable setuid and seggid (escalation)
- spawn: denies fork() and execve()
- resourcecontrol: disables process affinity and sched priority

@ - PROPOSITION 2' (comments)

> Note on QEMU escapes and Sandbox:
>
> Note that in the SECOND PROPOSITION being given in this document, we talk about having different PID and USER ID namespaces. The reason behind that is the sandbox feature still allows the QEMU processes to coexist the same USER ID namespace as other QEMU processes and, by having a different namespace, we would reduce the attack surface.
>
> Nevertheless, the fact that our example being used at this document already enables seccomp, disallows QEMU process to elevate privileges (become root), disallows QEMU process to fork/exec - so executing commands after an escalation is virtually impossible - and disallows QEMU to change scheduler decisions - trying to place the VM in a different CPU - is already a good measure to avoid QEMU escapes.

3.l-) QEMU logging: tracing execution not a good idea

The QEMU Machine Protocol (QMP) allows applications to operate a QEMU instance.

QMP is JSON based and features the following:

- Lightweight, text-based, easy to parse data format
- Asynchronous messages support (ie. events)
- Capabilities Negotiation

The HMP is the simple interactive monitor on QEMU, designed primarily for debugging and simple human use. Higher level tools should connect to the QMP which offers a stable interface with JSON to make it easy to parse reliably.

3.l.1-) QEMU Machine Protocol

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd help info

info balloon  -- show balloon information
info block [-n] [-v] [device] -- show info of one block device or all block devices (-n: show named nodes; -v: show
info block-jobs  -- show progress of ongoing block device operations
info blockstats  -- show block device statistics
info capture  -- show capture information
info chardev  -- show the character devices
info cpus  -- show infos for each CPU
info cpustats  -- show CPU statistics
info dump  -- Display the latest dump status
info history  -- show the command line history
info hotpluggable-cpus  -- Show information about hotpluggable CPUs
info ioapic  -- show io apic state
info iothreads  -- show iothreads
info irq  -- show the interrupts statistics (if available)
info jit  -- show dynamic compiler info
info KVM  -- show KVM information
info lapic [apic-id] -- show local apic state (apic-id: local apic to read, default is which of current CPU)
info mem  -- show the active virtual memory mappings
info memdev  -- show memory backends
info memory-devices  -- show memory devices
info memory_size_summary  -- show the amount of initially allocated and present hotpluggable (if enabled) me
info mice  -- show which guest mouse is receiving events
info migrate  -- show migration status
info migrate_cache_size  -- show current migration xbzrle cache size
info migrate_capabilities  -- show current migration capabilities
info migrate_parameters  -- show current migration parameters
info mtree [-f][-d][-o] -- show memory tree (-f: dump flat view for address spaces;-d: dump dispatch tree, valid w
info name  -- show the current VM name
info network  -- show the network state
info numa  -- show NUMA information
info opcount  -- show dynamic compiler opcode counters
info pci  -- show PCI info
info pic  -- show PIC state
info profile  -- show profiling information
info qdm  -- show qdev device model list
info qom-tree [path] -- show QOM composition tree
info qtree  -- show device tree
info ramblock  -- Display system ramblock information
info rdma  -- show RDMA state
info registers [-a] -- show the cpu registers (-a: all - show register info for all cpus)
info rocker name -- Show rocker switch
info rocker-of-dpa-flows name [tbl_id] -- Show rocker OF-DPA flow tables
info rocker-of-dpa-groups name [type] -- Show rocker OF-DPA groups
info rocker-ports name -- Show rocker ports
info roms  -- show roms
info sev  -- show SEV information
info snapshots  -- show the currently saved VM snapshots
info spice  -- show the spice server status
info status  -- show the current VM status (running|paused)
info sync-profile [-m] [-n] [max] -- show synchronization profiling info, up to max entries (default: 10), sorted by
info tlb  -- show virtual to physical memory mappings
info tpm  -- show the TPM device
info trace-events [name] [vcpu] -- show available trace-events & their state (name: event name pattern; vcpu: v
info usb  -- show guest USB devices
info usbhost  -- show host USB devices
info usernet  -- show user network stack connection states
info uuid  -- show the current VM UUID
info version  -- show the version of QEMU
info vm-generation-id  -- Show Virtual Machine Generation ID
info vnc  -- show the vnc server status
```

Example: getting information from a VM block device:

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd info block -v
libvirt-1-format: /var/lib/libvirt/images/bionic-disk01.qcow2 (qcow2)
    Attached to:      /machine/peripheral/virtio-disk0/virtio-backend
    Cache mode:       writeback

Images:
image: /var/lib/libvirt/images/bionic-disk01.qcow2
file format: qcow2
virtual size: 30 GiB (32212254720 bytes)
disk size: 19.3 GiB
cluster_size: 65536
Format specific information:
    compat: 0.10
    refcount bits: 16
```

Example: getting the VM memory tree information

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd info mtree -f -o
FlatView #0
 AS "memory", root: system
 AS "cpu-memory-0", root: system
 AS "cpu-memory-1", root: system
 AS "cpu-memory-2", root: system
 AS "cpu-memory-3", root: system
 AS "cpu-memory-4", root: system
 AS "cpu-memory-5", root: system
 AS "cpu-memory-6", root: system
 AS "cpu-memory-7", root: system
 AS "piix3-ide", root: bus master container
 AS "piix3-usb-uhci", root: bus master container
 AS "virtio-serial-pci", root: bus master container
 AS "virtio-blk-pci", root: bus master container
 AS "virtio-net-pci", root: bus master container
 AS "virtio-balloon-pci", root: bus master container
 Root memory region: system
  0000000000000000-00000000000bffff (prio 0, ram): pc.ram owner:{obj path=/objects/pc.ram} KVM
  00000000000c0000-00000000000c0fff (prio 0, rom): pc.ram @00000000000c0000 owner:{obj path=/objects/pc
  00000000000c1000-00000000000c3fff (prio 0, ram): pc.ram @00000000000c1000 owner:{obj path=/objects/pc
  00000000000c4000-00000000000e7fff (prio 0, rom): pc.ram @00000000000c4000 owner:{obj path=/objects/p
  00000000000e8000-00000000000effff (prio 0, ram): pc.ram @00000000000e8000 owner:{obj path=/objects/pc
  00000000000f0000-00000000000fffff (prio 0, rom): pc.ram @00000000000f0000 owner:{obj path=/objects/pc.
  0000000000100000-00000000bfffffff (prio 0, ram): pc.ram @0000000000100000 owner:{obj path=/objects/pc.
  00000000feb80000-00000000feb8002f (prio 0, i/o): msix-table owner:{dev id=net0}
  00000000feb80800-00000000feb80807 (prio 0, i/o): msix-pba owner:{dev id=net0}
  00000000feb81000-00000000feb8101f (prio 0, i/o): msix-table owner:{dev id=virtio-serial0}
  00000000feb81800-00000000feb81807 (prio 0, i/o): msix-pba owner:{dev id=virtio-serial0}
  00000000feb82000-00000000feb8201f (prio 0, i/o): msix-table owner:{dev id=virtio-disk0}
  00000000feb82800-00000000feb82807 (prio 0, i/o): msix-pba owner:{dev id=virtio-disk0}
  00000000feb83000-00000000feb8300f (prio 1, i/o): i6300esb owner:{dev id=watchdog0}
  00000000febf0000-00000000febf0fff (prio 0, i/o): virtio-pci-common owner:{dev id=net0}
  00000000febf1000-00000000febf1fff (prio 0, i/o): virtio-pci-isr owner:{dev id=net0}
  00000000febf2000-00000000febf2fff (prio 0, i/o): virtio-pci-device owner:{dev id=net0}
  00000000febf3000-00000000febf3fff (prio 0, i/o): virtio-pci-notify owner:{dev id=net0}
  00000000febf4000-00000000febf4fff (prio 0, i/o): virtio-pci-common owner:{dev id=virtio-serial0}
```

```
    00000000febf5000-00000000febf5fff (prio 0, i/o): virtio-pci-isr owner:{dev id=virtio-serial0}
    00000000febf6000-00000000febf6fff (prio 0, i/o): virtio-pci-device owner:{dev id=virtio-serial0}
    00000000febf7000-00000000febf7fff (prio 0, i/o): virtio-pci-notify owner:{dev id=virtio-serial0}
    00000000febf8000-00000000febf8fff (prio 0, i/o): virtio-pci-common owner:{dev id=virtio-disk0}
    00000000febf9000-00000000febf9fff (prio 0, i/o): virtio-pci-isr owner:{dev id=virtio-disk0}
    00000000febfa000-00000000febfafff (prio 0, i/o): virtio-pci-device owner:{dev id=virtio-disk0}
    00000000febfb000-00000000febfbfff (prio 0, i/o): virtio-pci-notify owner:{dev id=virtio-disk0}
    00000000febfc000-00000000febfcfff (prio 0, i/o): virtio-pci-common owner:{dev id=balloon0}
    00000000febfd000-00000000febfdfff (prio 0, i/o): virtio-pci-isr owner:{dev id=balloon0}
    00000000febfe000-00000000febfefff (prio 0, i/o): virtio-pci-device owner:{dev id=balloon0}
    00000000febff000-00000000febfffff (prio 0, i/o): virtio-pci-notify owner:{dev id=balloon0}
    00000000fec00000-00000000fec00fff (prio 0, i/o): KVM-ioapic owner:{dev path=/machine/i440fx/ioapic}
    00000000fed00000-00000000fed003ff (prio 0, i/o): hpet owner:{dev path=/machine/unattached/device[14]}
    00000000fee00000-00000000feefffff (prio 4096, i/o): KVM-apic-msi owner:{dev path=/machine/unattached/de
    00000000fffc0000-00000000ffffffff (prio 0, rom): pc.bios parent:{obj path=/machine/unattached} KVM
    0000000100000000-000000023fffffff (prio 0, ram): pc.ram @00000000c0000000 owner:{obj path=/objects/pc.r
    ...
```

Example: getting the QEMU virtual devices tree

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd info qtree
bus: main-system-bus
  type System
  dev: hpet, id ""
    gpio-in "" 2
    gpio-out "" 1
    gpio-out "sysbus-irq" 32
    timers = 3 (0x3)
    msi = false
    hpet-intcap = 4 (0x4)
    hpet-offset-saved = true
    mmio 00000000fed00000/0000000000000400
  dev: KVM-ioapic, id ""
    gpio-in "" 24
    gsi_base = 0 (0x0)
    mmio 00000000fec00000/0000000000001000
  dev: i440FX-pcihost, id ""
    pci-hole64-size = 2147483648 (2 GiB)
    short_root_bus = 0 (0x0)
    x-pci-hole64-fix = true
    bus: pci.0
      type PCI
      dev: virtio-balloon-pci, id "balloon0"
        disable-legacy = "off"
        disable-modern = false
        class = 255 (0xff)
        virtio-pci-bus-master-bug-migration = false
        migrate-extra = true
        modern-pio-notify = false
        x-disable-pcie = false
        page-per-vq = false
        x-ignore-backend-features = false
        ats = false
        x-pcie-deverr-init = true
        x-pcie-lnkctl-init = true
        x-pcie-pm-init = true
        x-pcie-flr-init = true
        addr = 05.0
        romfile = ""
        rombar = 1 (0x1)
        multifunction = false
        x-pcie-lnksta-dllla = true
        x-pcie-extcap-init = true
        failover_pair_id = ""
        class Class 00ff, addr 00:05.0, pci id 1af4:1002 (sub 1af4:0005)
        bar 0: i/o at 0xc100 [0xc11f]
        bar 4: mem at 0xfebfc000 [0xfebfffff]
        bus: virtio-bus
          type virtio-pci-bus
          dev: virtio-balloon-device, id ""
            deflate-on-oom = false
            free-page-hint = false
            QEMU-4-0-config-size = false
            indirect_desc = true
            event_idx = true
            notify_on_empty = true
            any_layout = true
            iommu_platform = false
            packed = false
            use-started = true
            use-disabled-flag = true
      dev: i6300esb, id "watchdog0"
        addr = 06.0
        romfile = ""
        rombar = 1 (0x1)
        multifunction = false
        x-pcie-lnksta-dllla = true
        x-pcie-extcap-init = true
        failover_pair_id = ""
        class Class 0880, addr 00:06.0, pci id 8086:25ab (sub 1af4:1100)
        bar 0: mem at 0xfeb83000 [0xfeb8300f]
      dev: virtio-net-pci, id "net0"
        disable-legacy = "off"
        disable-modern = false
        ioeventfd = true
        vectors = 3 (0x3)
        virtio-pci-bus-master-bug-migration = false
        migrate-extra = true
        modern-pio-notify = false
        x-disable-pcie = false
        page-per-vq = false
        x-ignore-backend-features = false
        ats = false
        x-pcie-deverr-init = true
        x-pcie-lnkctl-init = true
        x-pcie-pm-init = true
        x-pcie-flr-init = true
        addr = 02.0
        romfile = "efi-virtio.rom"
        rombar = 1 (0x1)
        multifunction = false
        ...
```

3.I.2-) QEMU QMP commands to change VM disks

This session shows how access to QMP **HAS TO BE SECURED**. By having access to VM QMP interface you can do pretty much anything regarding a VM administration lifecycle.

Example creating a QCOW2 file and adding to the VM:

```
$ sudo qemu-img create -f qcow2 /var/lib/libvirt/images/bionic-disk02.qcow2 5G
Formatting '/var/lib/libvirt/images/bionic-disk02.qcow2', fmt=qcow2 size=5368709120 cluster_size=65536 lazy_r

$ sudo chown -R libvirt-qemu: /var/lib/libvirt/images/bionic-disk02.qcow2
```

Adding a block device drive to the instance:

```
$ sudo virsh qemu-monitor-command --domain bionic --hmp --cmd drive_add 0 if=none,file=/var/lib/libvirt/ima
OK
```

Checking block device was added:

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd info block

libvirt-1-format: /var/lib/libvirt/images/bionic-disk01.qcow2 (qcow2)
    Attached to:      /machine/peripheral/virtio-disk0/virtio-backend
    Cache mode:       writeback

none0 (#block786): /var/lib/libvirt/images/bionic-disk02.qcow2 (qcow2)
    Removable device: not locked, tray closed
    Cache mode:       writeback
```

Adding a QEMU para-virtualized device to the guest:

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd device_add virtio-blk-pci,drive=none0,id=mydisk

$ virsh qemu-monitor-command --domain bionic --hmp --cmd info block
libvirt-1-format: /var/lib/libvirt/images/bionic-disk01.qcow2 (qcow2)
    Attached to:      /machine/peripheral/virtio-disk0/virtio-backend
    Cache mode:       writeback

none0 (#block786): /var/lib/libvirt/images/bionic-disk02.qcow2 (qcow2)
    Attached to:      /machine/peripheral/mydisk/virtio-backend
    Cache mode:       writeback
```

3.I.3-) QEMU Monitor Event Loop

Through the QMP interface, one is able to get SOME VM events (described HERE) such as:

```
SHUTDOWN
POWERDOWN
RESET
STOP
RESUME
SUSPEND
SUSPEND_DISK
WAKEUP
WATCHDOG
GUEST_PANICKED
GUEST_CRASHLOADED
MEMORY_FAILURE
JOB_STATUS_CHANGE
BLOCK_IMAGE_CORRUPTED
BLOCK_IO_ERROR
BLOCK_JOB_COMPLETED
BLOCK_JOB_CANCELLED
```

BLOCK_JOB_ERROR
BLOCK_JOB_READY
BLOCK_JOB_PENDING
BLOCK_WRITE_THRESHOLD
QUORUM_FAILURE
QUORUM_REPORT_BAD
DEVICE_TRAY_MOVED
PR_MANAGER_STATUS_CHANGED
BLOCK_EXPORT_DELETED
VSERPORT_CHANGE
DUMP_COMPLETED
NIC_RX_FILTER_CHANGED
FAILOVER_NEGOTIATED
RDMA_GID_STATUS_CHANGED
SPICE_CONNECTED
SPICE_INITIALIZED
SPICE_DISCONNECTED
SPICE_MIGRATE_COMPLETED
VNC_CONNECTED
VNC_INITIALIZED
VNC_DISCONNECTED
MIGRATION
MIGRATION_PASS
COLO_EXIT
UNPLUG_PRIMARY
DEVICE_DELETED
BALLOON_CHANGE
MEMORY_DEVICE_SIZE_CHANGE
MEM_UNPLUG_ERROR
RTC_CHANGE
ACPI_DEVICE_OST

Like the example bellow:

```
$ virsh qemu-monitor-event --loop --timestamp

2021-04-19 18:33:39.593+0000: event POWERDOWN for domain groovy: <null>
2021-04-19 18:33:40.512+0000: event RTC_CHANGE for domain _router: {"offset":1}
2021-04-19 18:33:41.367+0000: event SHUTDOWN for domain groovy: {"guest":true,"reason":"guest-shutdown"
2021-04-19 18:33:41.370+0000: event STOP for domain groovy: <null>
2021-04-19 18:33:41.370+0000: event SHUTDOWN for domain groovy: {"guest":false,"reason":"host-signal"}
2021-04-19 18:33:57.659+0000: event POWERDOWN for domain hirsute: <null>
2021-04-19 18:33:59.314+0000: event SHUTDOWN for domain hirsute: {"guest":true,"reason":"guest-shutdown'
2021-04-19 18:33:59.319+0000: event STOP for domain hirsute: <null>
2021-04-19 18:33:59.319+0000: event SHUTDOWN for domain hirsute: {"guest":false,"reason":"host-signal"}
2021-04-19 18:34:07.182+0000: event POWERDOWN for domain focal: <null>
2021-04-19 18:34:08.844+0000: event SHUTDOWN for domain focal: {"guest":true,"reason":"guest-shutdown"}
2021-04-19 18:34:08.849+0000: event STOP for domain focal: <null>
2021-04-19 18:34:08.849+0000: event SHUTDOWN for domain focal: {"guest":false,"reason":"host-signal"}
2021-04-19 18:34:18.350+0000: event POWERDOWN for domain bionic: <null>
2021-04-19 18:34:19.891+0000: event SHUTDOWN for domain bionic: {"guest":true,"reason":"guest-shutdown"}
2021-04-19 18:34:19.910+0000: event STOP for domain bionic: <null>
2021-04-19 18:34:19.913+0000: event VSERPORT_CHANGE for domain bionic: {"open":false,"id":"console1"}
2021-04-19 18:34:19.913+0000: event VSERPORT_CHANGE for domain bionic: {"open":false,"id":"console2"}
2021-04-19 18:34:19.913+0000: event VSERPORT_CHANGE for domain bionic: {"open":false,"id":"console3"}
2021-04-19 18:34:19.913+0000: event VSERPORT_CHANGE for domain bionic: {"open":false,"id":"console4"}
2021-04-19 18:34:19.913+0000: event VSERPORT_CHANGE for domain bionic: {"open":false,"id":"console5"}
2021-04-19 18:34:19.913+0000: event VSERPORT_CHANGE for domain bionic: {"open":false,"id":"console6"}
2021-04-19 18:34:19.921+0000: event RESET for domain bionic: {"guest":false,"reason":"host-qmp-system-reset
```

```
2021-04-19 18:34:19.922+0000: event RESUME for domain bionic: <null>
2021-04-19 18:34:19.950+0000: event RESET for domain bionic: {"guest":true,"reason":"guest-reset"}
2021-04-19 18:34:24.964+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console1"}
2021-04-19 18:34:24.964+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console2"}
2021-04-19 18:34:24.964+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console3"}
2021-04-19 18:34:24.964+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console4"}
2021-04-19 18:34:24.965+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console5"}
2021-04-19 18:34:24.965+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console6"}
2021-04-19 18:34:57.517+0000: event RTC_CHANGE for domain bionic: {"offset":1}
2021-04-19 18:44:53.526+0000: event RTC_CHANGE for domain _router: {"offset":1}
2021-04-19 18:46:13.509+0000: event RTC_CHANGE for domain bionic: {"offset":1}
2021-04-19 18:56:05.527+0000: event RTC_CHANGE for domain _router: {"offset":1}
2021-04-19 18:57:25.522+0000: event RTC_CHANGE for domain bionic: {"offset":1}
2021-04-19 19:07:17.527+0000: event RTC_CHANGE for domain _router: {"offset":1}
2021-04-19 19:08:37.527+0000: event RTC_CHANGE for domain bionic: {"offset":1}
2021-04-19 19:18:16.901+0000: event POWERDOWN for domain bionic: <null>
2021-04-19 19:18:18.429+0000: event SHUTDOWN for domain bionic: {"guest":true,"reason":"guest-shutdown"}
2021-04-19 19:18:18.435+0000: event STOP for domain bionic: <null>
2021-04-19 19:18:18.435+0000: event SHUTDOWN for domain bionic: {"guest":false,"reason":"host-signal"}
2021-04-19 19:18:27.515+0000: event RTC_CHANGE for domain _router: {"offset":1}
2021-04-19 19:18:47.875+0000: event RESUME for domain bionic: <null>
2021-04-19 19:18:52.389+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console1"}
2021-04-19 19:18:52.389+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console2"}
2021-04-19 19:18:52.389+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console3"}
2021-04-19 19:18:52.389+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console4"}
2021-04-19 19:18:52.389+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console5"}
2021-04-19 19:18:52.389+0000: event VSERPORT_CHANGE for domain bionic: {"open":true,"id":"console6"}
2021-04-19 19:18:54.511+0000: event NIC_RX_FILTER_CHANGED for domain bionic: {"name":"net0","path":"/ma
2021-04-19 19:19:25.518+0000: event RTC_CHANGE for domain bionic: {"offset":1}
2021-04-19 19:29:41.523+0000: event RTC_CHANGE for domain _router: {"offset":1}
event loop interrupted
events received: xxx
```

3.I.4-) QEMU LOGGING

Unfortunately logging QEMU is not a simple task. It is IMPOSSIBLE to try to log something useful if you don't know what to look for. Like the previous examople showed, despite the default EVENTS being sent through QMP interface, logging what is happening with a VM is not an easy task.

All the QEMU logging feature was more focused in its development side: If you know what QEMU internal subsystem to introspect it can be a VERY POWERFUL tool.

We will be focusing here how the TRACE feature can be used in order to LOG virtually ANYTHING from a QEMU VM instance running:

```
$ QEMU-system-x86_64 -d help
Log items (comma separated):
out_asm       show generated host assembly code for each compiled TB (translation block)
in_asm        show target assembly code for each compiled TB
op            show micro ops for each compiled TB
op_opt        show micro ops after optimization
op_ind        show micro ops before indirect lowering
int           show interrupts/exceptions in short format
exec          show trace before each executed TB (lots of logs)
cpu           show CPU registers before entering a TB (lots of logs)
fpu           include FPU registers in the 'cpu' logging
mmu           log MMU-related activities
pcall         x86 only: show protected mode far calls/returns/exceptions
cpu_reset     show CPU state before CPU resets
unimp         log unimplemented functionality
```

```
guest_errors    log when the guest OS does something invalid (eg accessing a non-existent register)
page            dump pages at beginning of user mode emulation
nochain         do not chain compiled TBs so that "exec" and "cpu" show complete traces
strace          log every user-mode syscall, its input, and its result
trace:PATTERN   enable trace events

Use "-d trace:help" to get a list of trace events.
```

Let's list all the available tracing events QEMU binary has (I'm grepping for ONLY the KVM related internal function/traces):

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd info trace-events | grep KVM_
KVM_sev_launch_finish : state 0
KVM_sev_launch_measurement : state 0
KVM_sev_launch_update_data : state 0
KVM_sev_launch_start : state 0
KVM_sev_change_state : state 0
KVM_memcrypt_unregister_region : state 0
KVM_memcrypt_register_region : state 0
KVM_sev_init : state 0
KVM_x86_update_msi_routes : state 0
KVM_x86_remove_msi_route : state 0
KVM_x86_add_msi_route : state 0
KVM_x86_fixup_msi_error : state 0
KVM_clear_dirty_log : state 0
KVM_set_user_memory : state 0
KVM_set_ioeventfd_pio : state 0
KVM_set_ioeventfd_mmio : state 0
KVM_irqchip_release_virq : state 0
KVM_irqchip_update_msi_route : state 0
KVM_irqchip_add_msi_route : state 0
KVM_irqchip_commit_routes : state 0
KVM_failed_reg_set : state 0
KVM_failed_reg_get : state 0
KVM_device_ioctl : state 0
KVM_run_exit : state 0
KVM_vcpu_ioctl : state 0
KVM_vm_ioctl : state 0
KVM_ioctl : state 0
```

Let's enable a single trace: **KVM_RUN_EXIT**. This function always run whenever a VM vCPU leaves the virtualization context and gives the CPU control back to the QEMU vCPU thread.

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd trace-event KVM_run_exit on
```

And check it was enabled:

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd info trace-events | grep KVM_
KVM_sev_launch_finish : state 0
KVM_sev_launch_measurement : state 0
KVM_sev_launch_update_data : state 0
KVM_sev_launch_start : state 0
KVM_sev_change_state : state 0
KVM_memcrypt_unregister_region : state 0
KVM_memcrypt_register_region : state 0
KVM_sev_init : state 0
KVM_x86_update_msi_routes : state 0
KVM_x86_remove_msi_route : state 0
```

```
KVM_x86_add_msi_route : state 0
KVM_x86_fixup_msi_error : state 0
KVM_clear_dirty_log : state 0
KVM_set_user_memory : state 0
KVM_set_ioeventfd_pio : state 0
KVM_set_ioeventfd_mmio : state 0
KVM_irqchip_release_virq : state 0
KVM_irqchip_update_msi_route : state 0
KVM_irqchip_add_msi_route : state 0
KVM_irqchip_commit_routes : state 0
KVM_failed_reg_set : state 0
KVM_failed_reg_get : state 0
KVM_device_ioctl : state 0
KVM_run_exit : state 1
KVM_vcpu_ioctl : state 0
KVM_vm_ioctl : state 0
KVM_ioctl : state 0
```

Now we are able to instruct QEMU, through the QMP interface, to LOG all times the traced event (of KVM_RUN_EXIT) happened. This will tell me all the times the vCPU "exited". It could have happened because of an DEVICE EMULATION need, or because an unhandled (during the virtualization context) interrupt has happened.

```
$ virsh qemu-monitor-command --domain bionic --hmp --cmd log trace:KVM_run_exit
```

And voilá, the QEMU log file will start showing, after the QEMU cmdline being used, the information we asked for:

```
2021-04-19 19:18:47.385+0000: starting up libvirt version: 6.6.0, package: 1ubuntu3.4 (Victor Manuel Tapia King
LC_ALL=C \
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin \
HOME=/var/lib/libvirt/QEMU/domain-14-bionic \
XDG_DATA_HOME=/var/lib/libvirt/QEMU/domain-14-bionic/.local/share \
XDG_CACHE_HOME=/var/lib/libvirt/QEMU/domain-14-bionic/.cache \
XDG_CONFIG_HOME=/var/lib/libvirt/QEMU/domain-14-bionic/.config \
QEMU_AUDIO_DRV=none \
/bin/QEMU-system-x86_64 \
-name guest=bionic,debug-threads=on \
-S \
-object secret,id=masterKey0,format=raw,file=/var/lib/libvirt/QEMU/domain-14-bionic/master-key.aes \
-machine pc-i440fx-5.0,accel=KVM,usb=off,dump-guest-core=off \
-cpu host,migratable=on \
-m 8192 \
-overcommit mem-lock=off \
-smp 8,sockets=8,cores=1,threads=1 \
-uuid 0cd4db6a-fcd0-4408-b271-61cd00155219 \
-display none \
-no-user-config \
-nodefaults \
-chardev socket,id=charmonitor,fd=36,server,nowait \
-mon chardev=charmonitor,id=monitor,mode=control \
-rtc base=utc,driftfix=slew \
-global KVM-pit.lost_tick_policy=delay \
-no-shutdown \
-global PIIX4_PM.disable_s3=1 \
-global PIIX4_PM.disable_s4=1 \
-boot strict=on \
-device piix3-usb-uhci,id=usb,bus=pci.0,addr=0x1.0x2 \
-device virtio-serial-pci,id=virtio-serial0,bus=pci.0,addr=0x3 \
-blockdev '{"driver":"file","filename":"/var/lib/libvirt/images/bionic-disk01.qcow2","node-name":"libvirt-1-storage
-blockdev '{"node-name":"libvirt-1-format","read-only":false,"driver":"qcow2","file":"libvirt-1-storage","backing":r
```

```
-device virtio-blk-pci,bus=pci.0,addr=0x4,drive=libvirt-1-format,id=virtio-disk0,bootindex=1 \
-netdev tap,fd=38,id=hostnet0,vhost=on,vhostfd=39 \
-device virtio-net-pci,netdev=hostnet0,id=net0,mac=52:54:00:4f:21:0a,bus=pci.0,addr=0x2 \
-chardev pty,id=charserial0 \
-device isa-serial,chardev=charserial0,id=serial0 \
-chardev pty,id=charconsole1 \
-device virtconsole,chardev=charconsole1,id=console1 \
-chardev pty,id=charconsole2 \
-device virtconsole,chardev=charconsole2,id=console2 \
-chardev pty,id=charconsole3 \
-device virtconsole,chardev=charconsole3,id=console3 \
-chardev pty,id=charconsole4 \
-device virtconsole,chardev=charconsole4,id=console4 \
-chardev pty,id=charconsole5 \
-device virtconsole,chardev=charconsole5,id=console5 \
-chardev pty,id=charconsole6 \
-device virtconsole,chardev=charconsole6,id=console6 \
-device i6300esb,id=watchdog0,bus=pci.0,addr=0x6 \
-watchdog-action inject-nmi \
-device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x5 \
-sandbox on,obsolete=deny,elevateprivileges=deny,spawn=deny,resourcecontrol=deny \
-msg timestamp=on
2021-04-19 19:18:47.385+0000: Domain id=14 is tainted: host-cpu
char device redirected to /dev/pts/23 (label charserial0)
char device redirected to /dev/pts/24 (label charconsole1)
char device redirected to /dev/pts/25 (label charconsole2)
char device redirected to /dev/pts/28 (label charconsole3)
char device redirected to /dev/pts/29 (label charconsole4)
char device redirected to /dev/pts/30 (label charconsole5)
char device redirected to /dev/pts/31 (label charconsole6)
2021-04-20 03:47:50.625+0000: Domain id=14 is tainted: custom-monitor
3572073@1618892213.515023:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515194:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515210:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515251:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515265:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515277:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515289:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515301:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515312:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515324:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515336:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515347:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515360:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515680:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515902:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515938:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515965:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515978:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.515990:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.516002:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.516014:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.516043:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.516066:KVM_run_exit cpu_index 4, reason 2
3572073@1618892213.516140:KVM_run_exit cpu_index 4, reason 6
3572073@1618892213.516224:KVM_run_exit cpu_index 4, reason 6
3572073@1618892213.516270:KVM_run_exit cpu_index 4, reason 2
```

And the reason why the vCPU has exited the virtualization context (reason 2 == IO emulation).

> As demonstrated in this session, QEMU logs can log pretty much anything happening in a virtual machine. The problem is not the data but what data, and what to do with the data. A more approachable way to logging could be to select some **key variables** from the tracing events and isolate them in a LOG (and consume this by some external tool if needed)

3.I.5-) LIBVIRT LOGGING

Libvirt provides logging - to QEMU VM instances - very similar to the QEMU QMP events described in previous item. It has also a fine-tuned logging mechanism that serves more for debug purposes.

Differently than with QEMU, no trace events need to be enabled. Instead, libvirt config file does specify what types of 'code sections' need to be enabled - to log - and the verbosity level of this logging.

You can also split different code sections logging into different log mechanisms - syslog, file, stderr - simultaneously.

```
log_filters="3:remote 4:event 3:util.json 3:rpc 1:*"
log_outputs="1:file:/var/log/libvirt/libvirtd.log"
```

If you target QEMU VMs debugging logging:

```
1:libvirt 1:QEMU 1:conf 1:security 3:event 3:json 3:file 3:object 1:util
```

If you want less verbose logging for QEMU VMs:

```
1:libvirt 1:QEMU 1:conf 1:security 3:event 3:json 3:file 3:object 1:util
```

or if you want a minimalistic QEMU QMP monitor logging:

```
2:QEMU.QEMU_monitor 3:*
```

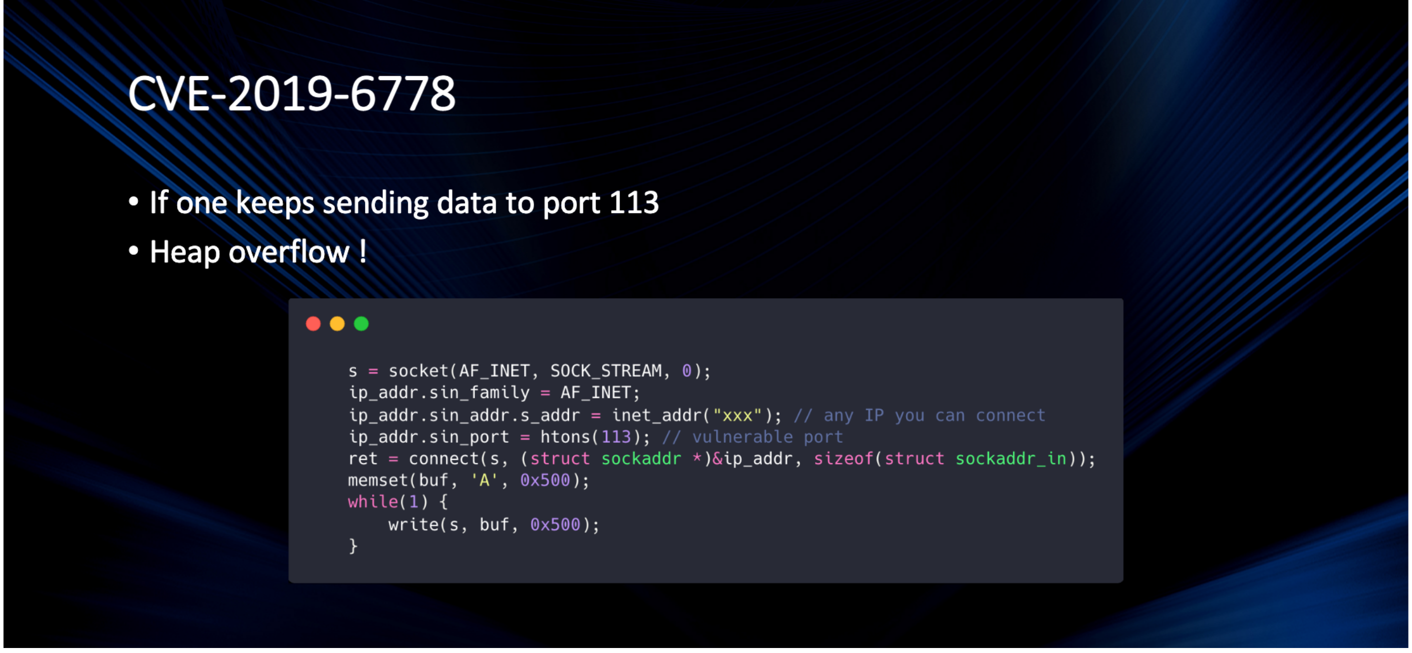
> As demonstrated in this session, as well as in previous session, Libvirt logs, just like QEMU logs, can log pretty much all there is. Again, the problem is not the data to extract from libvirt but what data, and what to do with the data. A more approachable way to logging could be to select some **key variables** from different internal libvirt sub-systems and isolate them in a LOG (and consume this by some external tool if needed)

4-) Learning with history: QEMU CVEs: study case

4.a-) CVE-2019-6778 - heap buffer overflow in SLIRP

Device emulation handled in **slirp_subr.c:tcp_emu**. Two important structures compromised: mbuf (store data from IP layer) and sbuf (store data frmo TCP layer).

If someone keeps sending data to an emulated SLIRP device at port 113 there might be a heap overflow:


CVE-2019-6778

- If one keeps sending data to port 113
- Heap overflow !

```
s = socket(AF_INET, SOCK_STREAM, 0);
ip_addr.sin_family = AF_INET;
ip_addr.sin_addr.s_addr = inet_addr("xxx"); // any IP you can connect
ip_addr.sin_port = htons(113); // vulnerable port
ret = connect(s, (struct sockaddr *)&ip_addr, sizeof(struct sockaddr_in));
memset(buf, 'A', 0x500);
while(1) {
    write(s, buf, 0x500);
}
```

The exploit takes advantage of the fact the QEMU SLIRP emulated device will allocate buffers for fragmented packets, in order to reconstruct a bigger packet at the end using the initial fragmented packet allocation buffer.

image29

By causing an overflow to an allocated data buffer, and combining requests for the host emulated device to deal with fragmented packets, one can 'steal' memory data from QEMU memory (at least of the part dealing with the device emulation).

image30

> This CVE shows a real possible attack to emulated devices - as long as you're using the SLIRP emulated device (also known as User Networking). Check **User Networking (SLIRP)** section in this same document.

4.b-) CVE-2019-14835 - V-gHost escape in virtio-net / vhost-net

V-gHost is a Linux kernel buffer overflow bug in the vhost kernel module which attackers can trigger from inside the VM when running with privileged accounts. One might think that is not a problem as QEMU VM processes don't run as privileged processes but that is not quite true during the live migration process, which is exactly what attackers could use together with this vulnerability.

A buffer overflow flaw was found, in versions from 2.6.34 to 5.2.x, in the way Linux kernel's vhost functionality - that translates virtqueue buffers to IOVs (I/O buffer vectors) - logged the buffer descriptors during migration. A privileged guest user, by using manipulated VIRTIO drivers, are able to pass descriptors with invalid length to the host when migration is underway and could use this flaw to either cause a denial of service OR even escalate privileges on the host.

> A proof-of-concept of this bug, resulting in host's kernel crash, can be seen HERE

> Because the nature of vhost-kernel, like explained in session QEMU Internals => Vhost protocol, all the techniques of confining the QEMU processes would have failed for this CVE, as part of the packet processing was offloaded to a specific kernel thread being executed on behalf of that particular guest.

5-) QEMU CVEs instrospection: a study case

5.a-) Lessons learned: avoid it happening again

Based on everything written in this document, it is reasonable to say that despite all efforts being put into hardening an hypervisor host OS, there will always be implementation details that might skip methods put in practice, allowing an attacker to be able to break something meaningful for other tenants. That is exactly the reason of this document: not to leverage the vulnerabilities or exploits, or try to explain how to detect them, but to better (and deeper) understand what is the attack surface and how attacks could be either mitigated or, if not, have its effects fully understood.

While attacks aiming the QEMU process might be mitigated by the sand-boxing features of seccomp and apparmor (or any other LSM), techniques being used by para-virtualization such as VIRTIO ring buffers shared with HostOS, or SR-IOV capabilities using IOMMU, might expose things out of the escope of the emulation layer (done by QEMU).

With that said, during the course of this document you will find items entitled: PROPOSITION 1, 2 and 3. Those items try to address different levels of attack surface in a way of what is needed for the environment used as example and a possible future environment.

1. PROPOSITION (1): Improve current LSM (apparmor) rules for the emulation layer (QEMU).
2. PROPOSITION (2): QEMU processes running with same user privileges: isolation levels to avoid other resources being jeopardized. This item was considered 'done' by the seccomp feature entirely as you may read in previous sections.
3. PROPOSITION (3): HostOS instrumentation for identifying, logging and blocking bloody-minded tenants. This topic will be described in the next section.

5.b-) Introspection tools: avoid overhead

Apart from confinement, a good strategy might be runtime security detection based on known pattern behaviors (from previous CVEs OR known security exploits techniques).

> For example: knowing what to look for could turn a detection system into something better than just logging OS blocked resources which could be, most of the time, false-positive alerts.

Introspection is a relatively new methodology for securing OSes and cloud-environments. As the time has passed, different tools allowed Operating Systems code to be probed and analyzed without - or with very little - overhead.

The different mechanisms to probe and introspect a running OS kernel always targeted tracing & profiling only. This has significantly changed lately.

Some of the observability, tracing and profiling tools are:

- Dtrace (born in Solaris 10)
- FreeBSD Dtrace (ported from Solaris)
- Systemtap (Linux version of a 'Dtrace-like' engine)
- Ftrace (Linux Function Tracing)
- Perf (Performance Counters Tool - Linux profiler tool)
- Valgrind (Instrumentation framework for dynamic analysis)
- ...

Without going into details about all them, instead, this document will try explain some of the key concepts that are currently being used for the security introspection subject.

Now, to the basics:

events

For the "Perf" tool, there are multiple hardware events - provided by latest architectures - to be used as data sources: cpu-cycles, instructions, cache-references, cache-misses, branch-instructions, branch-misses, bus-cycles, L1 data cache loads and stores, L1 instruction cache loads and stores, TLB loads and stores for data and instructions, etc. There are also multiple software events to be used: OS cpu-clock, task-clock, page-faults, minor-faults, major-faults, number of context seitches, cpu-migrations, etc.

Apart from that, within the kernel there are multiple specifically created tracing points to be used by either perf or ftrace-like tools. The tracing points are placed at scheduling events, interrupt handlers, memory-mapped I/O handlers, CPU power state transitions handlers, file-systems operators, virtualization handlers, and so.

Some of the times, the already existing kernel events are not enough for a tool to collect needed data for a business logic: events might not exist in the kernel parts of where you need to collect data from OR they don't provide the data you need. For that you may have to work with probes.

kprobes & uprobes

- **Kprobes**

Enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address, specifying a handler routine to be invoked when the breakpoint is hit. There are currently two types of probes: kprobes, and kretprobes (also called return probes). A kprobe can be inserted on virtually any instruction in the kernel. A return probe fires when a specified function returns.

In the typical case, Kprobes-based instrumentation is packaged as a kernel module. The module's init function registers one or more probes, and the exit function unregisters them. A registration function such as register_kprobe() specifies where the probe is to be inserted and what handler is to be called when the probe is hit.

- **Uprobes**

Uprobes enables you to dynamically break into any routine in a user application and collect debugging and performance information non-disruptively. You can trap at any code address, specifying a kernel handler routine to be invoked when the breakpoint is hit.

There are currently two types of user-space probes: uprobes and uretprobes (also called return probes). A uprobe can be inserted on any instruction in the application's virtual address space. A return probe fires when a specified user function returns. A registration function such as register_uprobe() specifies which process is to be probed, where the probe is to be inserted, and what handler is to be called when the probe is hit.

Typically, Uprobes-based instrumentation is packaged as a kernel module. In the simplest case, the module's init function registers one or more probes, and the exit function unregisters them. However, probes can be registered or unregistered in response to other events as well.

eBPF

> Finally this is where this session headed to. The eBPF technology is the technology that allows security introspection to happen in a modern way.

The Linux kernel has always been an ideal place to implement observability, networking, and security. Unfortunately this was often impractical as it required changing kernel source code or loading kernel modules, and resulted in layers of abstractions stacked on top of each other.

> **eBPF is a revolutionary technology that can run sandboxed programs in the Linux kernel without changing kernel source code or loading kernel modules.**

By making the Linux kernel programmable, infrastructure software can leverage existing layers, making them more intelligent and feature-rich without continuing to add additional layers of complexity to the system or compromising execution efficiency and safety.

eBPF has resulted in the development of a completely new generation of software able to reprogram the behavior of the Linux kernel and even apply logic across multiple subsystems which were traditionally completely independent.

What is eBPF being used for:

- **Security**

Building on the foundation of seeing and understanding all system calls and combining that with a packet and socket-level view of all networking operations allows for revolutionary new approaches to securing systems. While aspects of system call filtering, network-level filtering, and process context tracing have typically been handled by completely independent systems, eBPF allows for combining the visibility and control of all aspects to create security systems operating on more context with better level of control.

- Networking

The combination of programmability and efficiency makes eBPF a natural fit for all packet processing requirements of networking solutions. The programmability of eBPF enables adding additional protocol parsers and easily program any forwarding logic to meet changing requirements without ever leaving the packet processing context of the Linux kernel. The efficiency provided by the eBPF JIT compiler provides execution performance close to that of natively compiled in-kernel code.

- Tracing and Profiling

The ability to attach eBPF programs to trace points - explained previously - as well as kernel and user application probe points - also explained previously - allows unprecedented visibility into the runtime behavior of applications and the system itself. By giving introspection abilities to both the application and system side, both views can be combined, allowing powerful and unique insights to troubleshoot system performance problems. Advanced statistical data structures allow to extract meaningful visibility data in an efficient manner, without requiring the export of vast amounts of sampling data as typically done by similar systems.

- Observability & Monitoring

Instead of relying on static counters and gauges exposed by the operating system, eBPF enables the collection & in-kernel aggregation of custom metrics and generation of visibility events based on a wide range of possible sources. This extends the depth of visibility that can be achieved as well as reduces the overall system overhead significantly by only collecting the visibility data required and by generating histograms and similar data structures at the source of the event instead of relying on the export of samples.

> An example of portable eBPF code can be found HERE

5.c-) Near future: predict/log unwanted known behavior

Considering the 2 CVEs we described earlier:

- CVE-2019-6778 - heap buffer overflow in SLIRP
- CVE-2019-14835 - V-gHost escape in virtio-net / vhost-net

And what we have read so far about eBPF and introspection tools, the idea of this session is to explain how, by knowing the CVEs 'modus operandi', the HostOS could be monitored in a way that any attempt to exploit those vulnerabilities would be **at least** logged.

First we need to step back and talk about how the eBPF security tools work. I'll concentrate my efforts in a specific tool called tracee, just because it is an open-source tool and has a company, aquasecurity, providing support behind. This is also true for the tool falcosecurity, with the company sysdig behind, and similar functionality.

> Please do note that there are other security tools with similar functionality and that this document is tool-agnostic. Intent here is to demonstrate those tools purposes for introspection and NOT to be an advisor for which tool to be used.

Falco Security

Falco uses system calls to secure and monitor a system, by:

- Parsing the Linux system calls from the kernel at runtime
- Asserting the stream against a powerful rules engine
- Alerting when a rule is violated

it checks for:

- Privilege escalation using privileged containers
- Namespace changes using tools like setns
- Read/Writes to well-known directories
- Creating symlinks
- Ownership and Mode changes
- Unexpected network connections or socket mutations
- Spawned processes using execve
- Executing shell binaries such as sh, bash, csh, zsh, etc
- Executing SSH binaries such as ssh, scp, sftp, etc
- Mutating Linux coreutils executables
- Mutating login binaries

- Mutating shadowutil or passwd executables such as shadowconfig, pwck, chpasswd, getpasswd, change, useradd, etc, and others.

Falco rules:

Alerts are configurable downstream actions that can be as simple as logging to STDOUT or as complex as delivering a gRPC call to a client. Falco can send alerts to:

- stdout
- file
- Syslog
- spawned program
- HTTP[s] end point
- client through the gRPC API

Falco components:

- Userspace program: is the CLI tool falco that you can use to interact with Falco. The userspace program handles signals, parses information from a Falco driver, and sends alerts.
- Configuration: defines how Falco is run, what rules to assert, and how to perform alerts. For more information, see Configuration.
- Driver: is a software that adheres to the Falco driver specification and sends a stream of system call information. You cannot run Falco without installing a driver. Supported drivers:
    - (Default) Kernel module built on libscap and libsinsp C++ libraries
    - BPF probe built from the same modules
    - Userspace instrumentation

A falco rules example:

```
- list: my_programs
  items: [ls, cat, pwd]

- rule: my_programs_opened_file
  desc: track whenever a set of programs opens a file
  condition: proc.name in (my_programs) and evt.type=open
  output: a tracked program opened a file (user=%user.name command=%proc.cmdline file=%fd.name)
  priority: INFO
```

to detect when programs (ls, cat and pwd) have opened a file.

> Obviously example shown here is very simple. You will find some more examples HERE.

> Indeed much of the functionality in falcosecurity can be achieved by apparmor and other LSM modules (like SELinux) but in an easier, more dynamic and configurable way.

> Despite being a more mature project, I'll explain more about 'tracee project' as the falco event sources are mostly related to syscalls only, and interpretation of what they're doing in the OS.

Tracee

Now the reader might understand why I have chosen the 'tracee' tool as the one to give more details of for security purposes: Tracee tool, even being younger, seems to be more advanced into the eBPF area, covering more than just syscalls or specific events, it allows its own core to be extended by using eBPF libbpf based code, with its tracee-ebpf core, and to process rules after they have been collected, with its tracee-rules engine.

The project:

Tracee is a Runtime Security and forensics tool for Linux. It is using Linux eBPF technology to trace your system and applications at runtime, and analyze collected events to detect suspicious behavioral patterns. It is delivered as a Docker image that monitors the OS and detects suspicious behavior based on a pre-defined set of behavioral patterns.

In some cases, you might want to leverage Tracee's eBPF event collection capabilities directly, without involving the detection engine. This might be useful for debugging / troubleshooting / analysis / research / education. In this case you can run Tracee with the trace sub-command, which will start dumping raw data directly into standard output. There are many configurations and options available so you can control exactly what is being collected and how.

tracee components:

Tracee is composed of the following sub-projects, which are hosted in the aquasecurity/tracee git repository:

- Tracee-eBPF - Linux Tracing and Forensics using eBPF

Apart from syscalls being traced, tracee is also able to use kernel tracing events, kprobes, LSM hooks, XDP hooks (and possibly some other event sources). The current events that can be observed are:

Events:

```
System Calls:    Sets:                    Arguments:

read             [syscalls fs fs_read_write]          (int fd, void* buf, size_t count)
write            [syscalls fs fs_read_write]          (int fd, void* buf, size_t count)
open             [default syscalls fs fs_file_ops]    (const char* pathname, int flags, mode_t mode)
close            [default syscalls fs fs_file_ops]    (int fd)
stat             [default syscalls fs fs_file_attr]   (const char* pathname, struct stat* statbuf)
fstat            [default syscalls fs fs_file_attr]   (int fd, struct stat* statbuf)
lstat            [default syscalls fs fs_file_attr]   (const char* pathname, struct stat* statbuf)
poll             [syscalls fs fs_mux_io]              (struct pollfd* fds, unsigned int nfds, int timeout)
lseek            [syscalls fs fs_read_write]          (int fd, off_t offset, unsigned int whence)
mmap             [syscalls proc proc_mem]             (void* addr, size_t length, int prot, int flags, int fd, off_t off)
mprotect         [syscalls proc proc_mem]             (void* addr, size_t len, int prot)
munmap           [syscalls proc proc_mem]             (void* addr, size_t length)
brk              [syscalls proc proc_mem]             (void* addr)
rt_sigaction     [syscalls signals]                   (int signum, const struct sigaction* act, struct sigaction* oldact,
rt_sigprocmask   [syscalls signals]                   (int how, sigset_t* set, sigset_t* oldset, size_t sigsetsize)
rt_sigreturn     [syscalls signals]                   ()
ioctl            [syscalls fs fs_fd_ops]              (int fd, unsigned long request, unsigned long arg)
pread64          [syscalls fs fs_read_write]          (int fd, void* buf, size_t count, off_t offset)
pwrite64         [syscalls fs fs_read_write]          (int fd, const void* buf, size_t count, off_t offset)
readv            [syscalls fs fs_read_write]          (int fd, const struct iovec* iov, int iovcnt)
writev           [syscalls fs fs_read_write]          (int fd, const struct iovec* iov, int iovcnt)
access           [default syscalls fs fs_file_attr]   (const char* pathname, int mode)
pipe             [syscalls ipc ipc_pipe]              (int[2] pipefd)
select           [syscalls fs fs_mux_io]              (int nfds, fd_set* readfds, fd_set* writefds, fd_set* exceptfds, st
sched_yield      [syscalls proc proc_sched]           ()
mremap           [syscalls proc proc_mem]             (void* old_address, size_t old_size, size_t new_size, int flag
msync            [syscalls fs fs_sync]                (void* addr, size_t length, int flags)
mincore          [syscalls proc proc_mem]             (void* addr, size_t length, unsigned char* vec)
madvise          [syscalls proc proc_mem]             (void* addr, size_t length, int advice)
shmget           [syscalls ipc ipc_shm]               (key_t key, size_t size, int shmflg)
shmat            [syscalls ipc ipc_shm]               (int shmid, const void* shmaddr, int shmflg)
shmctl           [syscalls ipc ipc_shm]               (int shmid, int cmd, struct shmid_ds* buf)
dup              [default syscalls fs fs_fd_ops]      (int oldfd)
dup2             [default syscalls fs fs_fd_ops]      (int oldfd, int newfd)
pause            [syscalls signals]                   ()
nanosleep        [syscalls time time_timer]           (const struct timespec* req, struct timespec* rem)
getitimer        [syscalls time time_timer]           (int which, struct itimerval* curr_value)
```

alarm	[syscalls time time_timer]	(unsigned int seconds)
setitimer	[syscalls time time_timer]	(int which, struct itimerval* new_value, struct itimerval* old_
getpid	[syscalls proc proc_ids]	()
sendfile	[syscalls fs fs_read_write]	(int out_fd, int in_fd, off_t* offset, size_t count)
socket	[default syscalls net net_sock]	(int domain, int type, int protocol)
connect	[default syscalls net net_sock]	(int sockfd, struct sockaddr* addr, int addrlen)
accept	[default syscalls net net_sock]	(int sockfd, struct sockaddr* addr, int* addrlen)
sendto	[syscalls net net_snd_rcv]	(int sockfd, void* buf, size_t len, int flags, struct sockaddr* de
recvfrom	[syscalls net net_snd_rcv]	(int sockfd, void* buf, size_t len, int flags, struct sockaddr* s
sendmsg	[syscalls net net_snd_rcv]	(int sockfd, struct msghdr* msg, int flags)
recvmsg	[syscalls net net_snd_rcv]	(int sockfd, struct msghdr* msg, int flags)
shutdown	[syscalls net net_sock]	(int sockfd, int how)
bind	[default syscalls net net_sock]	(int sockfd, struct sockaddr* addr, int addrlen)
listen	[default syscalls net net_sock]	(int sockfd, int backlog)
getsockname	[default syscalls net net_sock]	(int sockfd, struct sockaddr* addr, int* addrlen)
getpeername	[syscalls net net_sock]	(int sockfd, struct sockaddr* addr, int* addrlen)
socketpair	[syscalls net net_sock]	(int domain, int type, int protocol, int[2] sv)
setsockopt	[syscalls net net_sock]	(int sockfd, int level, int optname, const void* optval, int optl
getsockopt	[syscalls net net_sock]	(int sockfd, int level, int optname, char* optval, int* optlen)
clone	[default syscalls proc proc_life]	(unsigned long flags, void* stack, int* parent_tid, int* child_t
fork	[default syscalls proc proc_life]	()
vfork	[default syscalls proc proc_life]	()
execve	[default syscalls proc proc_life]	(const char* pathname, const char*const* argv, const char
exit	[syscalls proc proc_life]	(int status)
wait4	[syscalls proc proc_life]	(pid_t pid, int* wstatus, int options, struct rusage* rusage)
kill	[default syscalls signals]	(pid_t pid, int sig)
uname	[syscalls system]	(struct utsname* buf)
semget	[syscalls ipc ipc_sem]	(key_t key, int nsems, int semflg)
semop	[syscalls ipc ipc_sem]	(int semid, struct sembuf* sops, size_t nsops)
semctl	[syscalls ipc ipc_sem]	(int semid, int semnum, int cmd, unsigned long arg)
shmdt	[syscalls ipc ipc_shm]	(const void* shmaddr)
msgget	[syscalls ipc ipc_msgq]	(key_t key, int msgflg)
msgsnd	[syscalls ipc ipc_msgq]	(int msqid, struct msgbuf* msgp, size_t msgsz, int msgflg)
msgrcv	[syscalls ipc ipc_msgq]	(int msqid, struct msgbuf* msgp, size_t msgsz, long msgtyp, ir
msgctl	[syscalls ipc ipc_msgq]	(int msqid, int cmd, struct msqid_ds* buf)
fcntl	[syscalls fs fs_fd_ops]	(int fd, int cmd, unsigned long arg)
flock	[syscalls fs fs_fd_ops]	(int fd, int operation)
fsync	[syscalls fs fs_sync]	(int fd)
fdatasync	[syscalls fs fs_sync]	(int fd)
truncate	[syscalls fs fs_file_ops]	(const char* path, off_t length)
ftruncate	[syscalls fs fs_file_ops]	(int fd, off_t length)
getdents	[default syscalls fs fs_dir_ops]	(int fd, struct linux_dirent* dirp, unsigned int count)
getcwd	[syscalls fs fs_dir_ops]	(char* buf, size_t size)
chdir	[syscalls fs fs_dir_ops]	(const char* path)
fchdir	[syscalls fs fs_dir_ops]	(int fd)
rename	[syscalls fs fs_file_ops]	(const char* oldpath, const char* newpath)
mkdir	[syscalls fs fs_dir_ops]	(const char* pathname, mode_t mode)
rmdir	[syscalls fs fs_dir_ops]	(const char* pathname)
creat	[default syscalls fs fs_file_ops]	(const char* pathname, mode_t mode)
link	[syscalls fs fs_link_ops]	(const char* oldpath, const char* newpath)
unlink	[default syscalls fs fs_link_ops]	(const char* pathname)
symlink	[default syscalls fs fs_link_ops]	(const char* target, const char* linkpath)
readlink	[syscalls fs fs_link_ops]	(const char* pathname, char* buf, size_t bufsiz)
chmod	[default syscalls fs fs_file_attr]	(const char* pathname, mode_t mode)
fchmod	[default syscalls fs fs_file_attr]	(int fd, mode_t mode)
chown	[default syscalls fs fs_file_attr]	(const char* pathname, uid_t owner, gid_t group)
fchown	[default syscalls fs fs_file_attr]	(int fd, uid_t owner, gid_t group)
lchown	[default syscalls fs fs_file_attr]	(const char* pathname, uid_t owner, gid_t group)
umask	[syscalls fs fs_file_attr]	(mode_t mask)
gettimeofday	[syscalls time time_tod]	(struct timeval* tv, struct timezone* tz)

getrlimit	[syscalls proc]	(int resource, struct rlimit* rlim)
getrusage	[syscalls proc]	(int who, struct rusage* usage)
sysinfo	[syscalls system]	(struct sysinfo* info)
times	[syscalls proc]	(struct tms* buf)
ptrace	[default syscalls proc]	(long request, pid_t pid, void* addr, void* data)
getuid	[syscalls proc proc_ids]	()
syslog	[syscalls system]	(int type, char* bufp, int len)
getgid	[syscalls proc proc_ids]	()
setuid	[default syscalls proc proc_ids]	(uid_t uid)
setgid	[default syscalls proc proc_ids]	(gid_t gid)
geteuid	[syscalls proc proc_ids]	()
getegid	[syscalls proc proc_ids]	()
setpgid	[syscalls proc proc_ids]	(pid_t pid, pid_t pgid)
getppid	[syscalls proc proc_ids]	()
getpgrp	[syscalls proc proc_ids]	()
setsid	[syscalls proc proc_ids]	()
setreuid	[default syscalls proc proc_ids]	(uid_t ruid, uid_t euid)
setregid	[default syscalls proc proc_ids]	(gid_t rgid, gid_t egid)
getgroups	[syscalls proc proc_ids]	(int size, gid_t* list)
setgroups	[syscalls proc proc_ids]	(int size, gid_t* list)
setresuid	[syscalls proc proc_ids]	(uid_t ruid, uid_t euid, uid_t suid)
getresuid	[syscalls proc proc_ids]	(uid_t* ruid, uid_t* euid, uid_t* suid)
setresgid	[syscalls proc proc_ids]	(gid_t rgid, gid_t egid, gid_t sgid)
getresgid	[syscalls proc proc_ids]	(gid_t* rgid, gid_t* egid, gid_t* sgid)
getpgid	[syscalls proc proc_ids]	(pid_t pid)
setfsuid	[default syscalls proc proc_ids]	(uid_t fsuid)
setfsgid	[default syscalls proc proc_ids]	(gid_t fsgid)
getsid	[syscalls proc proc_ids]	(pid_t pid)
capget	[syscalls proc]	(cap_user_header_t hdrp, cap_user_data_t datap)
capset	[syscalls proc]	(cap_user_header_t hdrp, const cap_user_data_t datap)
rt_sigpending	[syscalls signals]	(sigset_t* set, size_t sigsetsize)
rt_sigtimedwait	[syscalls signals]	(const sigset_t* set, siginfo_t* info, const struct timespec* tim
rt_sigqueueinfo	[syscalls signals]	(pid_t tgid, int sig, siginfo_t* info)
rt_sigsuspend	[syscalls signals]	(sigset_t* mask, size_t sigsetsize)
sigaltstack	[syscalls signals]	(const stack_t* ss, stack_t* old_ss)
utime	[syscalls fs fs_file_attr]	(const char* filename, const struct utimbuf* times)
mknod	[default syscalls fs fs_file_ops]	(const char* pathname, mode_t mode, dev_t dev)
uselib	[syscalls proc]	(const char* library)
personality	[syscalls system]	(unsigned long persona)
ustat	[syscalls fs fs_info]	(dev_t dev, struct ustat* ubuf)
statfs	[syscalls fs fs_info]	(const char* path, struct statfs* buf)
fstatfs	[syscalls fs fs_info]	(int fd, struct statfs* buf)
sysfs	[syscalls fs fs_info]	(int option)
getpriority	[syscalls proc proc_sched]	(int which, int who)
setpriority	[syscalls proc proc_sched]	(int which, int who, int prio)
sched_setparam	[syscalls proc proc_sched]	(pid_t pid, struct sched_param* param)
sched_getparam	[syscalls proc proc_sched]	(pid_t pid, struct sched_param* param)
sched_setscheduler	[syscalls proc proc_sched]	(pid_t pid, int policy, struct sched_param* param)
sched_getscheduler	[syscalls proc proc_sched]	(pid_t pid)
sched_get_priority_max	[syscalls proc proc_sched]	(int policy)
sched_get_priority_min	[syscalls proc proc_sched]	(int policy)
sched_rr_get_interval	[syscalls proc proc_sched]	(pid_t pid, struct timespec* tp)
mlock	[syscalls proc proc_mem]	(const void* addr, size_t len)
munlock	[syscalls proc proc_mem]	(const void* addr, size_t len)
mlockall	[syscalls proc proc_mem]	(int flags)
munlockall	[syscalls proc proc_mem]	()
vhangup	[syscalls system]	()
modify_ldt	[syscalls proc proc_mem]	(int func, void* ptr, unsigned long bytecount)
pivot_root	[syscalls fs]	(const char* new_root, const char* put_old)
sysctl	[syscalls system]	(struct __sysctl_args* args)

```
prctl           [default syscalls proc]           (int option, unsigned long arg2, unsigned long arg3, unsigned lor
arch_prctl          [syscalls proc]             (int option, unsigned long addr)
adjtimex            [syscalls time time_clock]           (struct timex* buf)
setrlimit          [syscalls proc]            (int resource, const struct rlimit* rlim)
chroot             [syscalls fs fs_dir_ops]           (const char* path)
sync              [syscalls fs fs_sync]            ()
acct             [syscalls system]              (const char* filename)
settimeofday          [syscalls time time_tod]           (const struct timeval* tv, const struct timezone* tz)
mount              [default syscalls fs]             (const char* source, const char* target, const char* filesystemty
```

```
sethostname          [syscalls net]                (const char* name, size_t len)
setdomainname          [syscalls net]                (const char* name, size_t len)
iopl              [syscalls system]              (int level)
ioperm             [syscalls system]              (unsigned long from, unsigned long num, int turn_on)
create_module         [syscalls system system_module]         ()
init_module          [default syscalls system system_module]  (void* module_image, unsigned long len, const cha
delete_module         [default syscalls system system_module]  (const char* name, int flags)
get_kernel_syms        [syscalls system system_module]         ()
query_module          [syscalls system system_module]         ()
quotactl            [syscalls system]              (int cmd, const char* special, int id, void* addr)
nfsservctl           [syscalls fs]               ()
getpmsg             [syscalls]                ()
putpmsg             [syscalls]                ()
afs               [syscalls]               ()
tuxcall             [syscalls]               ()
security             [syscalls]               ()
gettid             [syscalls proc proc_ids]           ()
readahead            [syscalls fs]               (int fd, off_t offset, size_t count)
setxattr           [syscalls fs fs_file_attr]          (const char* path, const char* name, const void* value, size_t siz
lsetxattr           [syscalls fs fs_file_attr]          (const char* path, const char* name, const void* value, size_t siz
fsetxattr           [syscalls fs fs_file_attr]          (int fd, const char* name, const void* value, size_t size, int flags)
getxattr           [syscalls fs fs_file_attr]          (const char* path, const char* name, void* value, size_t size)
lgetxattr           [syscalls fs fs_file_attr]          (const char* path, const char* name, void* value, size_t size)
fgetxattr           [syscalls fs fs_file_attr]          (int fd, const char* name, void* value, size_t size)
listxattr          [syscalls fs fs_file_attr]          (const char* path, char* list, size_t size)
llistxattr          [syscalls fs fs_file_attr]          (const char* path, char* list, size_t size)
flistxattr          [syscalls fs fs_file_attr]          (int fd, char* list, size_t size)
removexattr          [syscalls fs fs_file_attr]           (const char* path, const char* name)
lremovexattr          [syscalls fs fs_file_attr]           (const char* path, const char* name)
fremovexattr          [syscalls fs fs_file_attr]           (int fd, const char* name)
tkill             [syscalls signals]             (int tid, int sig)
time              [syscalls time time_tod]            (time_t* tloc)
futex              [syscalls ipc ipc_futex]           (int* uaddr, int futex_op, int val, const struct timespec* timeout,
sched_setaffinity      [syscalls proc proc_sched]           (pid_t pid, size_t cpusetsize, unsigned long* mask)
sched_getaffinity      [syscalls proc proc_sched]           (pid_t pid, size_t cpusetsize, unsigned long* mask)
set_thread_area       [syscalls proc]              (struct user_desc* u_info)
io_setup           [syscalls fs fs_async_io]           (unsigned int nr_events, io_context_t* ctx_idp)
io_destroy           [syscalls fs fs_async_io]           (io_context_t ctx_id)
io_getevents          [syscalls fs fs_async_io]           (io_context_t ctx_id, long min_nr, long nr, struct io_event* ev
io_submit           [syscalls fs fs_async_io]           (io_context_t ctx_id, long nr, struct iocb** iocbpp)
io_cancel           [syscalls fs fs_async_io]           (io_context_t ctx_id, struct iocb* iocb, struct io_event* result)
get_thread_area       [syscalls proc]              (struct user_desc* u_info)
lookup_dcookie        [syscalls fs fs_dir_ops]           (u64 cookie, char* buffer, size_t len)
epoll_create          [syscalls fs fs_mux_io]           (int size)
epoll_ctl_old         [syscalls fs fs_mux_io]           ()
epoll_wait_old        [syscalls fs fs_mux_io]           ()
remap_file_pages       [syscalls]                (void* addr, size_t size, int prot, size_t pgoff, int flags)
```

getdents64	[default syscalls fs fs_dir_ops]	(unsigned int fd, struct linux_dirent64* dirp, unsigned int
set_tid_address	[syscalls proc]	(int* tidptr)
restart_syscall	[syscalls signals]	()
semtimedop	[syscalls ipc ipc_sem]	(int semid, struct sembuf* sops, size_t nsops, const struct ti
fadvise64	[syscalls fs]	(int fd, off_t offset, size_t len, int advice)
timer_create	[syscalls time time_timer]	(const clockid_t clockid, struct sigevent* sevp, timer_t* tim
timer_settime	[syscalls time time_timer]	(timer_t timer_id, int flags, const struct itimerspec* new_v
timer_gettime	[syscalls time time_timer]	(timer_t timer_id, struct itimerspec* curr_value)
timer_getoverrun	[syscalls time time_timer]	(timer_t timer_id)
timer_delete	[syscalls time time_timer]	(timer_t timer_id)
clock_settime	[syscalls time time_clock]	(const clockid_t clockid, const struct timespec* tp)
clock_gettime	[syscalls time time_clock]	(const clockid_t clockid, struct timespec* tp)
clock_getres	[syscalls time time_clock]	(const clockid_t clockid, struct timespec* res)
clock_nanosleep	[syscalls time time_clock]	(const clockid_t clockid, int flags, const struct timespec*
exit_group	[syscalls proc proc_life]	(int status)
epoll_wait	[syscalls fs fs_mux_io]	(int epfd, struct epoll_event* events, int maxevents, int timeou
epoll_ctl	[syscalls fs fs_mux_io]	(int epfd, int op, int fd, struct epoll_event* event)
tgkill	[syscalls signals]	(int tgid, int tid, int sig)
utimes	[syscalls fs fs_file_attr]	(char* filename, struct timeval* times)
vserver	[syscalls]	()
mbind	[syscalls system system_numa]	(void* addr, unsigned long len, int mode, const unsigned
set_mempolicy	[syscalls system system_numa]	(int mode, const unsigned long* nodemask, unsigne
get_mempolicy	[syscalls system system_numa]	(int* mode, unsigned long* nodemask, unsigned lon
mq_open	[syscalls ipc ipc_msgq]	(const char* name, int oflag, mode_t mode, struct mq_attr*
mq_unlink	[syscalls ipc ipc_msgq]	(const char* name)
mq_timedsend	[syscalls ipc ipc_msgq]	(mqd_t mqdes, const char* msg_ptr, size_t msg_len, unsi
mq_timedreceive	[syscalls ipc ipc_msgq]	(mqd_t mqdes, char* msg_ptr, size_t msg_len, unsigned
mq_notify	[syscalls ipc ipc_msgq]	(mqd_t mqdes, const struct sigevent* sevp)
mq_getsetattr	[syscalls ipc ipc_msgq]	(mqd_t mqdes, const struct mq_attr* newattr, struct mq_at
kexec_load	[syscalls system]	(unsigned long entry, unsigned long nr_segments, struct kexec_
waitid	[syscalls proc proc_life]	(int idtype, pid_t id, struct siginfo* infop, int options, struct rusa
d_key	[syscalls system system_keys]	(const char* type, const char* description, const void* pa
request_key	[syscalls system system_keys]	(const char* type, const char* description, const char*
keyctl	[syscalls system system_keys]	(int operation, unsigned long arg2, unsigned long arg3, unsi
ioprio_set	[syscalls proc proc_sched]	(int which, int who, int ioprio)
ioprio_get	[syscalls proc proc_sched]	(int which, int who)
inotify_init	[syscalls fs fs_monitor]	()
inotify_add_watch	[syscalls fs fs_monitor]	(int fd, const char* pathname, u32 mask)
inotify_rm_watch	[syscalls fs fs_monitor]	(int fd, int wd)
migrate_pages	[syscalls system system_numa]	(int pid, unsigned long maxnode, const unsigned lon
openat	[default syscalls fs fs_file_ops]	(int dirfd, const char* pathname, int flags, mode_t mode)
mkdirat	[syscalls fs fs_dir_ops]	(int dirfd, const char* pathname, mode_t mode)
mknodat	[default syscalls fs fs_file_ops]	(int dirfd, const char* pathname, mode_t mode, dev_t dev)
fchownat	[default syscalls fs fs_file_attr]	(int dirfd, const char* pathname, uid_t owner, gid_t group, i
futimesat	[syscalls fs fs_file_attr]	(int dirfd, const char* pathname, struct timeval* times)
newfstatat	[syscalls fs fs_file_attr]	(int dirfd, const char* pathname, struct stat* statbuf, int flags)
unlinkat	[default syscalls fs fs_link_ops]	(int dirfd, const char* pathname, int flags)
renameat	[syscalls fs fs_file_ops]	(int olddirfd, const char* oldpath, int newdirfd, const char* ne
linkat	[syscalls fs fs_link_ops]	(int olddirfd, const char* oldpath, int newdirfd, const char* new
symlinkat	[default syscalls fs fs_link_ops]	(const char* target, int newdirfd, const char* linkpath)
readlinkat	[syscalls fs fs_link_ops]	(int dirfd, const char* pathname, char* buf, int bufsiz)
fchmodat	[default syscalls fs fs_file_attr]	(int dirfd, const char* pathname, mode_t mode, int flags)
faccessat	[default syscalls fs fs_file_attr]	(int dirfd, const char* pathname, int mode, int flags)
pselect6	[syscalls fs fs_mux_io]	(int nfds, fd_set* readfds, fd_set* writefds, fd_set* exceptfds, s
ppoll	[syscalls fs fs_mux_io]	(struct pollfd* fds, unsigned int nfds, struct timespec* tmo_p, co
unshare	[syscalls proc]	(int flags)
set_robust_list	[syscalls ipc ipc_futex]	(struct robust_list_head* head, size_t len)
get_robust_list	[syscalls ipc ipc_futex]	(int pid, struct robust_list_head** head_ptr, size_t* len_ptr)
splice	[syscalls ipc ipc_pipe]	(int fd_in, off_t* off_in, int fd_out, off_t* off_out, size_t len, unsig
tee	[syscalls ipc ipc_pipe]	(int fd_in, int fd_out, size_t len, unsigned int flags)

sync_file_range	[syscalls fs fs_sync]	(int fd, off_t offset, off_t nbytes, unsigned int flags)
vmsplice	[syscalls ipc ipc_pipe]	(int fd, const struct iovec* iov, unsigned long nr_segs, unsigne
move_pages	[syscalls system system_numa]	(int pid, unsigned long count, const void** pages, con
utimensat	[syscalls fs fs_file_attr]	(int dirfd, const char* pathname, struct timespec* times, int fla
epoll_pwait	[syscalls fs fs_mux_io]	(int epfd, struct epoll_event* events, int maxevents, int timeo
signalfd	[syscalls signals]	(int fd, sigset_t* mask, int flags)
timerfd_create	[syscalls time time_timer]	(int clockid, int flags)
eventfd	[syscalls signals]	(unsigned int initval, int flags)
fallocate	[syscalls fs fs_file_ops]	(int fd, int mode, off_t offset, off_t len)
timerfd_settime	[syscalls time time_timer]	(int fd, int flags, const struct itimerspec* new_value, stru
timerfd_gettime	[syscalls time time_timer]	(int fd, struct itimerspec* curr_value)
accept4	[default syscalls net net_sock]	(int sockfd, struct sockaddr* addr, int* addrlen, int flags)
signalfd4	[syscalls signals]	(int fd, const sigset_t* mask, size_t sizemask, int flags)
eventfd2	[syscalls signals]	(unsigned int initval, int flags)
epoll_create1	[syscalls fs fs_mux_io]	(int flags)
dup3	[default syscalls fs fs_fd_ops]	(int oldfd, int newfd, int flags)
pipe2	[syscalls ipc ipc_pipe]	(int* pipefd, int flags)
inotify_init1	[syscalls fs fs_monitor]	(int flags)
preadv	[syscalls fs fs_read_write]	(int fd, const struct iovec* iov, unsigned long iovcnt, unsigned
pwritev	[syscalls fs fs_read_write]	(int fd, const struct iovec* iov, unsigned long iovcnt, unsigned
rt_tgsigqueueinfo	[syscalls signals]	(pid_t tgid, pid_t tid, int sig, siginfo_t* info)
perf_event_open	[syscalls system]	(struct perf_event_attr* attr, pid_t pid, int cpu, int group_fd,
recvmmsg	[syscalls net net_snd_rcv]	(int sockfd, struct mmsghdr* msgvec, unsigned int vlen, int
fanotify_init	[syscalls fs fs_monitor]	(unsigned int flags, unsigned int event_f_flags)
fanotify_mark	[syscalls fs fs_monitor]	(int fanotify_fd, unsigned int flags, u64 mask, int dirfd, const
prlimit64	[syscalls proc]	(pid_t pid, int resource, const struct rlimit64* new_limit, struct rlin
name_to_handle_at	[syscalls fs fs_file_ops]	(int dirfd, const char* pathname, struct file_handle* hanc
open_by_handle_at	[syscalls fs fs_file_ops]	(int mount_fd, struct file_handle* handle, int flags)
clock_adjtime	[syscalls time time_clock]	(const clockid_t clk_id, struct timex* buf)
syncfs	[syscalls fs fs_sync]	(int fd)
sendmmsg	[syscalls net net_snd_rcv]	(int sockfd, struct mmsghdr* msgvec, unsigned int vlen, in
setns	[syscalls proc]	(int fd, int nstype)
getcpu	[syscalls system system_numa]	(unsigned int* cpu, unsigned int* node, struct getcpu_ca
process_vm_readv	[default syscalls proc]	(pid_t pid, const struct iovec* local_iov, unsigned long liov
process_vm_writev	[default syscalls proc]	(pid_t pid, const struct iovec* local_iov, unsigned long liov
kcmp	[syscalls proc]	(pid_t pid1, pid_t pid2, int type, unsigned long idx1, unsigned long
finit_module	[default syscalls system system_module]	(int fd, const char* param_values, int flags)
sched_setattr	[syscalls proc proc_sched]	(pid_t pid, struct sched_attr* attr, unsigned int flags)
sched_getattr	[syscalls proc proc_sched]	(pid_t pid, struct sched_attr* attr, unsigned int size, unsigi
renameat2	[syscalls fs fs_file_ops]	(int olddirfd, const char* oldpath, int newdirfd, const char* n
seccomp	[syscalls proc]	(unsigned int operation, unsigned int flags, const void* args)
getrandom	[syscalls fs]	(void* buf, size_t buflen, unsigned int flags)
memfd_create	[default syscalls fs fs_file_ops]	(const char* name, unsigned int flags)
kexec_file_load	[syscalls system]	(int kernel_fd, int initrd_fd, unsigned long cmdline_len, const c
bpf	[default syscalls system]	(int cmd, union bpf_attr* attr, unsigned int size)
execveat	[default syscalls proc proc_life]	(int dirfd, const char* pathname, const char*const* argv, c
userfaultfd	[syscalls system]	(int flags)
membarrier	[syscalls proc proc_mem]	(int cmd, int flags)
mlock2	[syscalls proc proc_mem]	(const void* addr, size_t len, int flags)
copy_file_range	[syscalls fs fs_read_write]	(int fd_in, off_t* off_in, int fd_out, off_t* off_out, size_t len,
preadv2	[syscalls fs fs_read_write]	(int fd, const struct iovec* iov, unsigned long iovcnt, unsignec
pwritev2	[syscalls fs fs_read_write]	(int fd, const struct iovec* iov, unsigned long iovcnt, unsignec
pkey_mprotect	[default syscalls proc proc_mem]	(void* addr, size_t len, int prot, int pkey)
pkey_alloc	[syscalls proc proc_mem]	(unsigned int flags, unsigned long access_rights)
pkey_free	[syscalls proc proc_mem]	(int pkey)
statx	[syscalls fs fs_file_attr]	(int dirfd, const char* pathname, int flags, unsigned int mask, stru
io_pgetevents	[syscalls fs fs_async_io]	(aio_context_t ctx_id, long min_nr, long nr, struct io_event* e
rseq	[syscalls]	(struct rseq* rseq, u32 rseq_len, int flags, u32 sig)
pidfd_send_signal	[syscalls signals]	(int pidfd, int sig, siginfo_t* info, unsigned int flags)
io_uring_setup	[syscalls]	(unsigned int entries, struct io_uring_params* p)

```
io_uring_enter        [syscalls]                        (unsigned int fd, unsigned int to_submit, unsigned int min_compl
io_uring_register     [syscalls]                        (unsigned int fd, unsigned int opcode, void* arg, unsigned int nr_
open_tree             [syscalls]                        (int dfd, const char* filename, unsigned int flags)
move_mount            [default syscalls fs]             (int from_dfd, const char* from_path, int to_dfd, const char*
fsopen                [syscalls fs]                     (const char* fsname, unsigned int flags)
fsconfig              [syscalls fs]                     (int* fs_fd, unsigned int cmd, const char* key, const void* value, int
fsmount               [syscalls fs]                     (int fsfd, unsigned int flags, unsigned int ms_flags)
fspick                [syscalls fs]                     (int dirfd, const char* pathname, unsigned int flags)
pidfd_open            [syscalls]                        (pid_t pid, unsigned int flags)
clone3                [default syscalls proc proc_life]    (struct clone_args* cl_args, size_t size)
close_range           [default syscalls fs fs_file_ops]    (unsigned int first, unsigned int last)
openat2               [default syscalls fs fs_file_ops]    (int dirfd, const char* pathname, struct open_how* how, siz
pidfd_getfd           [syscalls]                        (int pidfd, int targetfd, unsigned int flags)
faccessat2            [default syscalls fs fs_file_attr]   (int fd, const char* path, int mode, int flag)
process_madvise       [syscalls]                        (int pidfd, void* addr, size_t length, int advice, unsigned long fla
epoll_pwait2          [syscalls fs fs_mux_io]           (int fd, struct epoll_event* events, int maxevents, const struc
```

```
Other Events:       Sets:                       Arguments:
_____________       _____                       ______________

sys_enter             []                        (int syscall)
sys_exit              []                        (int syscall)
do_exit               [proc proc_life]          ()
cap_capable           [default]                 (int cap, int syscall)
security_bprm_check   [default lsm_hooks]       (const char* pathname, dev_t dev, unsigned long inode)
security_file_open    [default lsm_hooks]       (const char* pathname, int flags, dev_t dev, unsigned long
security_inode_unlink [default lsm_hooks]       (const char* pathname)
vfs_write             []                        (const char* pathname, dev_t dev, unsigned long inode, size_t count, of
vfs_writev            []                        (const char* pathname, dev_t dev, unsigned long inode, unsigned long
mem_prot_alert        []                        (alert_t alert)
sched_process_exit    [default proc proc_life]  ()
commit_creds          []                        (int old_euid, int new_euid, int old_egid, int new_egid, int old_fsuid, i
switch_task_ns        []                        (pid_t pid, u32 new_mnt, u32 new_pid, u32 new_uts, u32 new_ipc, u3
magic_write           []                        (const char* pathname, bytes bytes)
security_socket_create [lsm_hooks]              (int family, int type, int protocol, int kern)
security_socket_listen [lsm_hooks]              (int sockfd, struct sockaddr* local_addr, int backlog)
security_socket_connect [lsm_hooks]             (int sockfd, struct sockaddr* remote_addr)
security_socket_accept [lsm_hooks]              (int sockfd, struct sockaddr* local_addr)
security_socket_bind   [lsm_hooks]              (int sockfd, struct sockaddr* local_addr)
security_sb_mount     [default lsm_hooks]       (const char* dev_name, const char* path, const char* ty
security_bpf          [lsm_hooks]               (int cmd)
security_bpf_map      [lsm_hooks]               (unsigned int map_id, const char* map_name)
```

Tracing

You can either opt to trace all registered events or select a few ones, including filters for processes and/or specific conditions. These options can be given directly at the command line in the 'collecting' phase.

Example:

> Let's trace a 'ls' command, started in the Host (not in a container), from uid 1000, only for new pids (from now on) and all possible registered events and arguments:

```
$ sudo ./dist/tracee-ebpf --trace comm=ls --trace '!container' --trace 'uid=1000' --trace pid=new
TIME(s)        UID    COMM           PID      TID      RET         EVENT              ARGS
387170.180056  1000   ls             3103403 3103403 -2            access             pathname: /etc/ld.so.preload, mo
387170.180115  1000   ls             3103403 3103403 0             security_file_open   pathname: /etc/ld.so.cache, fl
387170.180151  1000   ls             3103403 3103403 3             openat             dirfd: -100, pathname: /etc/ld.so.c
387170.180171  1000   ls             3103403 3103403 0             fstat              fd: 3, statbuf: 0x7FFC73471DA0
```

```
387170.180203  1000   ls            3103403 3103403 0           close              fd: 3
387170.180237  1000   ls            3103403 3103403 0           security_file_open   pathname: /usr/lib/x86_64-lin
387170.180261  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /lib/x86_64
387170.180291  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73471DF0
387170.180405  1000   ls            3103403 3103403 0           close              fd: 3
387170.180451  1000   ls            3103403 3103403 0           security_file_open   pathname: /usr/lib/x86_64-lin
387170.180490  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /lib/x86_64
387170.180519  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73471DD0
387170.180608  1000   ls            3103403 3103403 0           close              fd: 3
387170.180632  1000   ls            3103403 3103403 0           security_file_open   pathname: /usr/lib/x86_64-lin
387170.180645  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /lib/x86_64
387170.180660  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73471DB0
387170.180714  1000   ls            3103403 3103403 0           close              fd: 3
387170.180732  1000   ls            3103403 3103403 0           security_file_open   pathname: /usr/lib/x86_64-lin
387170.180745  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /lib/x86_64
387170.180759  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73471D90
387170.180907  1000   ls            3103403 3103403 0           close              fd: 3
387170.180934  1000   ls            3103403 3103403 0           security_file_open   pathname: /usr/lib/x86_64-lin
387170.180968  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /lib/x86_64
387170.181009  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73471D50
387170.181106  1000   ls            3103403 3103403 0           close              fd: 3
387170.181587  1000   ls            3103403 3103403 0           security_file_open   pathname: /proc/filesystems,
387170.181642  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /proc/filesy
387170.181677  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73472A50
387170.181726  1000   ls            3103403 3103403 0           close              fd: 3
387170.181749  1000   ls            3103403 3103403 -2          access             pathname: /etc/selinux/config, mo
387170.181814  1000   ls            3103403 3103403 0           security_file_open   pathname: /usr/lib/locale/loca
387170.181831  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /usr/lib/loc
387170.181841  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FCCDA1957A0
387170.181877  1000   ls            3103403 3103403 0           close              fd: 3
387170.182021  1000   ls            3103403 3103403 0           security_file_open   pathname: /, flags: O_RDONL
387170.182046  1000   ls            3103403 3103403 3           openat             dirfd: -100, pathname: /, flags: O_
387170.182062  1000   ls            3103403 3103403 0           fstat             fd: 3, statbuf: 0x7FFC73472750
387170.182120  1000   ls            3103403 3103403 752          getdents64          fd: 3, dirp: 0x563B7EA4AA30, c
387170.182157  1000   ls            3103403 3103403 0           getdents64          fd: 3, dirp: 0x563B7EA4AA30, co
387170.182169  1000   ls            3103403 3103403 0           close              fd: 3
387170.182238  1000   ls            3103403 3103403 0           fstat             fd: 1, statbuf: 0x7FFC73470620
387170.182307  1000   ls            3103403 3103403 0           close              fd: 1
387170.182342  1000   ls            3103403 3103403 0           close              fd: 2
387170.182558  1000   ls            3103403 3103403 0           sched_process_exit

End of events stream
Stats: {eventCounter:44 errorCounter:0 lostEvCounter:0 lostWrCounter:0}
```

As you can see, this looks very close to what a 'strace' output would look like. Nevertheless, the core engine for tracee allows for customized filters, not necessarily based in syscalls, that will tell whether a set of events and arguments happened... and THAT is what might warn you about security breakage.

> Like for example a tenant trying to access a file it should not, or trying to memory map address areas it should not, trying to create device node files it should not, etc.

One might ask what is the difference from that and apparmor, or other LSM engines. The difference is that LSM hooks are just ONE kind of events that might be supported in tracee. Other events might be created based on SPECIFIC needs, like internal-to-kernel access attempts, etc.

Another example, tracee is able to warn if another application has tampered itself and its internal ebpf kernel structures, by having 2 events coming from the LSM hooks inside the kernel.

> Like explained before, tracee has an eBPF core to introspect the kernel, and has maps shared among its userland and kernel ebpf code. If a ill-intentioned person tries to alter its internal structures, to remove or change the list of events to be monitored, for example, it can monitor itself by observing kernel functions responsible for dealing with those requests:

```
$ sudo ./dist/tracee-ebpf --debug --trace comm=bpftool --trace event=security_bpf_map
found bpf object file at: /tmp/tracee/tracee.bpf.5_8_0-43-generic.v0_5_1-24-g22a3058.o
TIME(s)        UID    COMM             PID      TID      RET              EVENT                ARGS
387505.754238  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 935, map_name: ar
387505.754556  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 936, map_name: bi
387505.754729  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 937, map_name: bu
387505.754902  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 938, map_name: bu
387505.755072  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 939, map_name: ch
387505.755240  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 940, map_name: co
387505.755411  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 941, map_name: co
387505.755588  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 942, map_name: ev
387505.755762  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 943, map_name: fil
387505.755937  0      bpftool          3149471 3149471 0                security_bpf_map     map_id: 944, map_name: fil
```

> Here the bpftool (userland tool) is reading all existing eBPF maps within the running kernel. Tracee was able to realize that it read its own eBPF maps and would be able to tell you if its internal structures were 'tainted' or 'tampared' by a 3rd party program.

This is just one example.

For the an Host OS, one could create RULES based on previous CVEs behavior that would monitor the OS kernel for security breakage attempts based on known/existent security issues (CVEs)

OR

Generic security-related 'recipes' (called 'signatures') could be created based in 'generic ways of security breakage attempt'.

SO,

Both approaches would allow HostOS admins to be warned about bad cloud citizens to could be better investigated or even blocked from the service.

> Okay, we've collected all this events data. And now ?

- Tracee-Rules - Runtime Security Detection Engine

Tracee supports authoring rules in Golang or in Rego (the language of Open Policy Agent). See tracee-rules rego examples for example Rego signatures, or tracee-rules golang examples

As explained in previous section, there are some generic security 'signatures' that could be created in order to avoid malicious code to be executed.

> One good example is when attacker tries to use LD_LIBRARY_PATH ordering to hijack library calls from a binary and execute arbitrary code. A good recipe made in 'REGO' can be found HERE.

@ - PROPOSITION 3

> **THIS IS THE THIRD PROPOSITION OF THIS DOCUMENT STEP**
>
> As a medium/long term approach, introspection tools should be analyzed for the HostOS environment. HostOS kernel should provide means for the next-generation security analysis tools to be available to run in a way that no impact is made (like eBPF support with CO.RE feature - compile once, run everywhere - and the BTF debugging format support - available in most recent kernels).
>
> With time, recipes (or signatures, in case of tracee) could be created in order to log security breakage attempts based on known CVEs and known access or execution patterns.

This will definitely be more effective than trying to 'log' a common set of existing events in order to guarantee the environment safety.

6-) REFERENCE

https://en.wikipedia.org/wiki/QEMU
https://wiki.QEMU.org/Documentation/TCG
https://wiki.QEMU.org/Features/
https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine
https://vfio.blogspot.com/2014/08/iommu-groups-inside-and-out.html
https://en.wikipedia.org/wiki/Input%E2%80%93output_memory_management_unit
https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net
https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net
https://www.redhat.com/en/blog/achieving-network-wirespeed-open-standard-manner-introducing-vdpa
https://www.programmersought.com/article/84854610738/
https://www.slideshare.net/janghoonsim/KVM-performance-optimization-for-ubuntu
https://www.redhat.com/en/blog/journey-vhost-users-realm
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_a
https://wiki.ubuntu.com/rafaeldtinoco?action=AttachFile&do=view&target=QEMU_vuln_and_mit_explained.htm
https://en.wikipedia.org/wiki/Kernel_page-table_isolation
https://www.programmersought.com/article/30564621193/
https://software.intel.com/security-software-guidance/resources/processors-affected-special-register-buffer-da
https://software.intel.com/security-software-guidance/deep-dives/deep-dive-special-register-buffer-data-sampl
https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-transactional-synchronizatio
https://www.QEMU.org/2021/01/19/virtio-blk-scsi-configuration/
https://en.wikipedia.org/wiki/LIO_(SCSI_target)
https://en.wikipedia.org/wiki/Ceph_(software)
https://www.programmersought.com/article/29323748848/
https://QEMU.readthedocs.io/en/latest/system/security.htm
https://readthedocs.org/projects/QEMU/downloads/pdf/latest/
https://en.wikipedia.org/wiki/AppArmor
https://gitlab.com/apparmor/apparmor/-/wikis/Libvirt
https://blade.tencent.com/en/advisories/v-ghost/
https://github.com/0xKira/QEMU-vm-escape
Tensec2019-Vulnerability_Discovery_and_Exploitation_of_Virtualization_Solutions_for_Cloud_Computing_and_D
Intel Doc: PCI-SIG SR-IOV Primer (Introd to SR-IOV Technology)
https://libvirt.org/cgroups.html
https://en.wikipedia.org/wiki/Linux_namespaces
https://en.wikipedia.org/wiki/Seccomp
https://vmsplice.net/~stefan/stefanha-KVM-forum-2018.pdf
https://wiki.QEMU.org/Documentation/QMP
https://libvirt.org/kbase/debuglogs.html#turning-on-debug-logs
https://github.com/0xKira/QEMU-vm-escape/blob/master/Tensec2019-Vulnerability_Discovery_and_Exploitatio
https://wiki.QEMU.org/Documentation/Networking
https://www.openwall.com/lists/oss-security/2019/09/17/1
https://www.kernel.org/doc/Documentation/kprobes.txt
https://github.com/jav/systemtap/blob/master/runtime/uprobes/uprobes.txt
https://perf.wiki.kernel.org/index.php/Tutorial
https://ebpf.io/
https://github.com/aquasecurity/libbpfgo
https://github.com/libbpf/libbpf/
https://aquasecurity.github.io/tracee/v0.5.1/rules-authoring/
https://github.com/falcosecurity/falco